

Lernen heißt, ein Feuer zu entfachen, und heißt nicht, Fässer zu befüllen.



Skript

Datenbank- systeme

Eine 'Education 2.0' Vorlesung

Autor: Prof. Dr. Paul F. Kirchberg
Version: 2.1
Datum: 23. September 2021

Dieses Skript zur Vorlesung "Datenbanksysteme" erhebt, wie die meisten Skripte, nicht den Anspruch Bücher zu ersetzen. Bücher sind sorgfältiger geschrieben und meist auch ausführlicher. Insbesondere ersetzt dieses Skript nicht eine aufmerksame Teilnahme an der Vorlesung.

Zur Erstellung dieses Skripts wurden folgende Quellen herangezogen:

- ▶ Vossen, G.: Datenbankmodelle, Datenbanksprachen und Datenbankmanagementsysteme
- ▶ Kemper, A.; Eickler, A.: Datenbanksysteme – Eine Einführung
- ▶ Härder, T.; Rahm, E.: Datenbanksysteme – Konzepte und Techniken der Implementierung
- ▶ IBM DB2 Benutzerhandbücher
- ▶ ORACLE Benutzerhandbücher
- ▶ Heuer, A.; Saake, G.: Datenbanken: Konzepte und Sprachen
- ▶ Zehnder, C.A.: Informationssysteme und Datenbanken
- ▶ Conrad, Stefan: Föderierte Datenbanksysteme
- ▶ Mutschler,Bela: Specht, Günter: Mobile Datenbanksysteme
- ▶ Meier, Andreas: Wüst, Thomas: Objektorientierte und objektrelationale Datenbanken
- ▶ Meier, Andreas: Relationale und postrelationale Datenbanken (nicht empfehlenswert)
- ▶ Geppert, Andreas: Objektrelationale und objektorientierte Datenbankkonzepte und -systeme
- ▶ Hohenstein; Lauffer; Schmatz; Weikert: Objektorientierte Datenbanksysteme
- ▶ Heinz Burnus: Datenbankentwicklung in IT-Berufen
- ▶ Prof. Dr. Alfons Kemper, Dr. Marin Wimmer: Übungsbuch Datenbanksysteme
- ▶ Andreas Meier: Relationale und postrelationale Datenbanken
- ▶ Unterstein, Michael; Matthiessen, Günter: Anwendungsentwicklung mit Datenbanken
- ▶ E. Rahm; G. Saake; K.-U. Sattler: Verteiltes und paralleles Datenmanagement
- ▶ Andreas Meier, Michael Kaufmann: SQL & NoSQL-Datenbanken
- ▶ Leonid Nossov, Hanno Ernst, Victor Chupis: Formales SQL-Tuning für Oracle-Datenbanken

Alle in diesem Text genannten Gebrauchsnamen, Handelsnamen, Marken, Produktnamen, usw. unterliegen warenzeichen-, marken- oder patentrechtlichem Schutz bzw. sind Warenzeichen oder eingetragene Warenzeichen der jeweiligen Inhaber. Die Wiedergabe solcher Namen und Bezeichnungen in diesem Text berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Gesetzgebung zu Warenzeichen und Markenschutz als frei zu betrachten wären und daher von jedermann benutzt werden dürfen.

Inhaltsverzeichnis

1 Überblick	1
1.1 Datenhaltung in Dateien	1
1.2 Datenunabhängigkeit	2
1.3 Datenbanksysteme	3
1.4 Entwicklungsgeschichte	6
1.5 Anforderungen an ein Datenbanksystem	7
2 Datenbankentwurf mit dem Entity-Relationship-Modell	9
2.1 Phasen des Entwurfsprozesses	9
2.2 Allgemeine Abstraktionskonzepte	11
2.3 Das Entity-Relationship-Modell	12
2.4 Konzeptioneller Entwurf mit dem ER-Modell	24
2.5 Ergänzungsaufgaben	25
3 Das relationale Datenmodell	27
3.1 Relationen und Relationenschemata	27
3.2 Modellinhärente Integritätsbedingungen	29
3.3 Die zwölf RDBMS-Regeln	32
3.4 Überführung von ER-Diagrammen in das Relationenmodell	33
4 Relationale Entwurfstheorie	41
4.1 Merkmale schlechter Relationenschemata	41
4.2 Funktionale Abhängigkeiten	42
4.3 Normalformen	43
5 Die Sprache SQL	53
5.1 Demodatenbanken	54
5.2 Datendefinition	54
5.3 Datenbankanfragen	62
5.4 Rekursion in SQL3	78
5.5 Datenmanipulationen	78
5.6 Transaktionsbefehle	81
5.7 Temporäre Tabellen	85
5.8 Systemkatalog	85
5.9 Datenkontrolle	87
5.10 Anwendungsentwicklung	99
6 Objektrelationale Konzepte	103
6.1 Kollektionen	104
6.2 Zeilentypen	106
6.3 Strukturierte Datentypen - Typkonstruktoren	107
6.4 Referenzspalten	112

6.5 View-Hierarchien	113
6.6 DISTINCT Datentypen	114
6.7 Objektrelationale Möglichkeiten bei DB2	116
6.8 Objektrelationale Möglichkeiten bei Oracle	117
7 Architektur von Datenbanksystemen	119
7.1 Die ANSI/SPARC-Schemaebenen	120
7.2 Komponenten eines Datenbanksystems	121
8 Zugriffspfade	125
8.1 Primärindizes	127
8.2 Sekundärindizes	134
8.3 Bewertung eindimensionaler Zugriffspfade	135
8.4 Tabellenübergreifende Zugriffspfade	136
8.5 Mehrdimensionale Zugriffspfade	138
8.6 Weitere Verfahren	142
9 Synchronisation	145
9.1 Anomalien im Mehrbenutzerbetrieb	145
9.2 Korrektheitskriterium der Serialisierbarkeit	148
9.3 Überblick zu Synchronisationsverfahren	149
9.4 Grundlagen von Sperrverfahren	150
9.5 Konsistenzstufen	158
9.6 Optimistische Synchronisation	160
9.7 Leistungsbewertung von Synchronisationsverfahren	162
10 Datenbankrecovery	165
10.1 Fehler- und Recovery-Arten	165
10.2 Logging-Techniken	168
10.3 Abhängigkeiten zur Pufferverwaltung	169
10.4 Sicherungspunkte	172
10.5 Aufbau der Log-Datei	173
10.6 Crash-Recovery	174
10.7 Geräte-Recovery	175
11 Physische Datenorganisation	177
11.1 Einsatz von Speichermedien	177
11.2 Dateien	178
11.3 DB-Pufferverwaltung	180
11.4 Satzverwaltung	185
A Das Datenbank-Trainingssystem	189
A.1 Einführung in das Datenbanktrainingssystem DBTS	189
A.2 Einrichten des DBTS	189
A.3 Die Übungsoberfläche im Überblick	190
A.4 Erläuterungen der SQL Abfragen	191
A.5 Das Datenbankmodell	192
A.6 Die Wiederherstellung der Datenbank	192
B Darstellungskonventionen	193

Kapitel 1

Überblick

Daten sind allgegenwärtig. Sie dienen als Grundlage von Entscheidungen und beeinflussen unser Handeln. Hierzu werden Daten vielfältig erfasst und elektronisch gespeichert. Für ein im Bedarfsfall schnelles Auffinden benötigter Daten ist es nützlich, wenn das Abspeichern der Daten strukturiert und geordnet erfolgt. Dies kann man über selbst definierte Dateien (mit oder ohne unterstützende Formate beispielsweise durch Tabellenkalkulationsprogramme) oder Datenbanksysteme umsetzen. Der Einsatz von Datenbanksystemen hat hierbei eine Reihe von Vorteilen. Dieses Skript verdeutlicht die Aufgaben und Möglichkeiten beim Einsatz von Datenbanksystemen und zeigt deren Programmierung sowie Optimierungsmöglichkeiten beim Einsatz.

1.1 Datenhaltung in Dateien

Die Speicherung der für eine Anwendung notwendigen Daten kann in vom Betriebssystem verwalteten Dateien erfolgen, so dass sich die Frage stellt, wofür man ein Datenbanksystem benötigt. So bieten Betriebssysteme mit den zugehörigen Dateisystemen eine Reihe von Funktionen für

- ▶ Erzeugung / Löschen von Dateien
- ▶ Zugriffsmöglichkeiten auf Blöcke/Sätze der Datei
- ▶ Einfache Operationen zum Lesen / Ändern / Löschen / Einfügen von Sätzen
- ▶ Verwaltung von Dateiverzeichnissen
- ▶ Vergabe von Zugriffsrechten
- ▶ Komprimierung / Verschlüsselung

Wesentlich bei der Verwendung von Dateien ist die statische Zuordnung von Daten zu den Programmen, die diese verarbeiten. Greift ein Programm auf eine Datei zu, so muss der Dateiaufbau dem Programm bekannt sein, die Strukturen müssen im Programm entsprechend definiert werden. Die Speicherungsstrukturen sind damit in die Programme eingebettet. Jedes Programm enthält damit eine eigene 'Datenbeschreibung', welche ausschließlich und direkt auf die Struktur einer durch das Programm zu bearbeitenden Datei abgestimmt ist. Eine dateibasierte Lösung bringt weiterhin folgende Probleme:

- ▶ Inflexibilität gegenüber Veränderungen in der Anwendung. Falls sich Informationselemente bei einer Dateiart ergeben, sind diese nur mit großem Aufwand realisierbar. Die Programme haben eine sehr geringe 'Datenunabhängigkeit' (vgl. Abschnitt 1.2).

- ▶ Hohe Redundanz aufgrund der Mehrfachspeicherung. Redundanz entsteht häufig dadurch, dass Anwendungsprogramme und die dazugehörigen Dateien zu unterschiedlichen Zeiten von verschiedenen Programmierern ohne hinreichende Abstimmung erstellt werden, so dass gemeinsam verwendete Daten unentdeckt bleiben können.
- ▶ Gefahr der Inkonsistenz durch die Möglichkeit, dass einzelne Programme an den von ihnen benutzten Dateien Veränderungen vornehmen, ohne dass diese Änderungen auch gleichzeitig von allen anderen Programmen vorgenommen werden, die die gleichen Daten nutzen. So kann sich beispielsweise in der Personaldatei der Name eines Lesers aufgrund einer Heirat ändern, ohne dass diese Änderung auch in der Projektdatei angepasst wird.
- ▶ Datenschutzprobleme können dadurch entstehen, dass der Zugriff auf die Dateien nicht immer angemessen kontrolliert werden kann. Ein erlaubter Zugriff auf Teile einer Datei ist nur schwierig realisierbar.
- ▶ Die effiziente Suche in großen Datenmengen ist nur mit aufwändigen Algorithmen und Datenstrukturen realisierbar.
- ▶ Die Zugriffsmöglichkeiten sind beschränkt. Es ist sehr aufwändig die in isolierten Dateien abgelegten Daten miteinander zu verknüpfen, d.h. Informationen aus einer Datei mit anderen logisch verwandten Daten aus einer anderen Datei zu verknüpfen.
- ▶ Wenn Daten in isolierten Dateien gehalten werden, wird die Wiederherstellung eines konsistenten Zustands der Gesamtinformationsmenge im Fehlerfall sehr schwierig. Im Allgemeinen bieten Dateisysteme bestenfalls die Möglichkeit einer periodisch durchgeföhrten Sicherung der Dateien.
- ▶ Je nach Anwendungsgebiet gibt es vielfältige Einschränkungen bezüglich der möglichen Werte, die so genannten Integritätsbedingungen. Dies beginnt bei einfachen Beschränkungen durch die Angabe eines Datentyps bis hin zu komplexeren Erfassungs- und Änderungsbedingungen. So darf beispielsweise ein Student eine Prüfung nur einmal wiederholen. Die Einhaltung derartiger Integritätsbedingungen ist bei der isolierten Speicherung der Informationseinheiten in verschiedenen Dateien sehr schwierig, da man zur Kontrolle Daten aus unterschiedlichen Dateien verknüpfen muss.
- ▶ Die Entwicklungskosten sind sehr hoch. In vielen Fällen müssen für die Entwicklung eines neuen Anwendungsprogramms die Grundaufgaben der Datenhaltung neu in das Projekt aufgenommen und implementiert werden.

1.2 Datenunabhängigkeit

Programme können mehr oder weniger von den physischen Datenspeicherung abhängen. Je unabhängiger die Programme von der Datenhaltung sind, desto größer ist dabei die Datenunabhängigkeit, die sich in fünf Stufen einteilen lässt.

- 1. Keine Datenunabhängigkeit:** Die Programme greifen direkt mittels Speicheradressen auf einen Datenträger zu. Dies hat eine Anpassung der Programme zur Folge, wenn die Position der Daten auf dem Datenträger sich ändert. Diese Stufe der Datenunabhängigkeit findet sich meist nur innerhalb von Betriebssystemen. So wird beispielsweise die Freispeicherliste eines Datenträgers an festen Positionen gespeichert.
- 2. Physische Datenunabhängigkeit:** Programme nutzen hier Funktionen des Betriebssystems um auf Daten und Dateien zuzugreifen. Hierdurch sind für das Programm keine Informationen über den Speicherort notwendig, es reicht die Angabe von Dateien und Verzeichnissen, Betriebssystem setzt dies in Lokalitätsinformationen um. Dies erlaubt auch ein Verschieben

der Datei auf dem Datenträger ohne dass das Programm neu übersetzt werden muss. Der Aufbau der Daten innerhalb der Datei muss dem Programm jedoch bekannt sein. So muss die Reihenfolge der Werte innerhalb der Datei im Programm bekannt und implementiert sein. Ändert sich der Dateiaufbau und damit die logische Struktur der Datei, muss das Programm angepasst werden.

3. **Teil-logische Datenunabhängigkeit:** Hierunter fallen Programme, die auf mehreren Dateiarten ohne spezielle Anpassungen arbeiten können. So gibt es beispielsweise Sortierprogramme, die beliebige Textdateien nach bestimmten Kriterien sortieren. Die Kriterien sowie Grundinformationen zu der Datei sind bei diesem Grad an Datenunabhängigkeit als Aufrufparameter mitzugeben.
4. **Logische und physische Datenunabhängigkeit:** Hier werden die Daten zusammen mit Informationen zu dem Speicherformat abgelegt. Der Datenaufbau ist damit nicht mehr Bestandteil des Anwendungsprogramms. Dieser Grad an Datenunabhängigkeit wird über Datenbanken erreicht. Auf die Daten wird über eine Zugriffssprache zugegriffen, die keine Informationen über die Speicherung benötigt. Änderungen an den Datenstrukturen, wie beispielsweise die Hinzunahme neuer Attribute, erfordern dadurch keine Anpassung der Programme. Auch wenn ein Attribut entfernt wird, sind alle Programme, die dieses Attribut nicht nutzen, weiterhin lauffähig.
5. **Geographische Datenunabhängigkeit:** Daten können an unterschiedlichen Standorten abgelegt werden. Bei Einsatz verteilter Datenbanksysteme bleibt diese Verteilung dem Benutzer verborgen. Wird auf Daten zugegriffen, die nicht lokal vorhanden sind, kommunizieren die Datenbankprogramme auf den Servern untereinander, so dass die gewünschten Daten zum Endbenutzer gelangen, ohne dass dieser erfährt, woher die Daten kommen. Die verschiedenen Standorte erscheinen wie eine große einheitliche Datenbank.

1.3 Datenbanksysteme

Datenbanksysteme unterstützen die Datenunabhängigkeit und übernehmen dabei eine ganze Reihe von Aufgaben, um die Probleme, die bei Dateilösungen entstehen können, zu lösen. Datenbanksysteme sind dabei ein Hilfsmittel zur effizienten, rechnergestützten Organisation, Erzeugung, Manipulation und Verwaltung großer Datensammlungen. Anwendungsprogramme senden hierzu Datenanforderungen an das Datenbanksystem, das die Operationen mit den gewünschten Daten durchführt und das Anwendungsprogramm über den Verlauf der Operationen informiert und ihm bei Bedarf die gewünschten Daten zur Verfügung stellt. Dabei können drei Ebenen unterschieden werden.

Die Anwendung beinhaltet die vom Benutzer gewünschte Kernfunktionalität wie beispielsweise Buchhaltungsfunktionen in einer Finanzbuchhaltungssoftware.

Das Datenbankmanagementsystem (DBMS) (auch Datenbankverwaltungssystem (DBVS)) stellt die Datenverwaltungsfunktionen wie Speichern und Suchen von Daten zur Verfügung. Dazu gehören auch Maßnahmen zur Zugriffskontrolle und Steuerung des Mehrbenutzerbetriebs.

Die Datenbank enthält die zu verwaltenden Daten.

Der Datenbestand (bzw. die Datenbank) sowie das Datenbankverwaltungssystem bilden zusammen ein Datenbanksystem. Dies ist wiederum Teil eines *Informationssystems*. Unter diesem Oberbegriff werden alle Systeme zusammengefasst, welche Informationen über eine bestimmte Anwendung bzw. Außenweltsituation speichern und verwalten können. Sie gestatten dabei das 'Arbeiten' mit diesen Informationen nach ausgewählten Gesichtspunkten und Zielsetzungen.

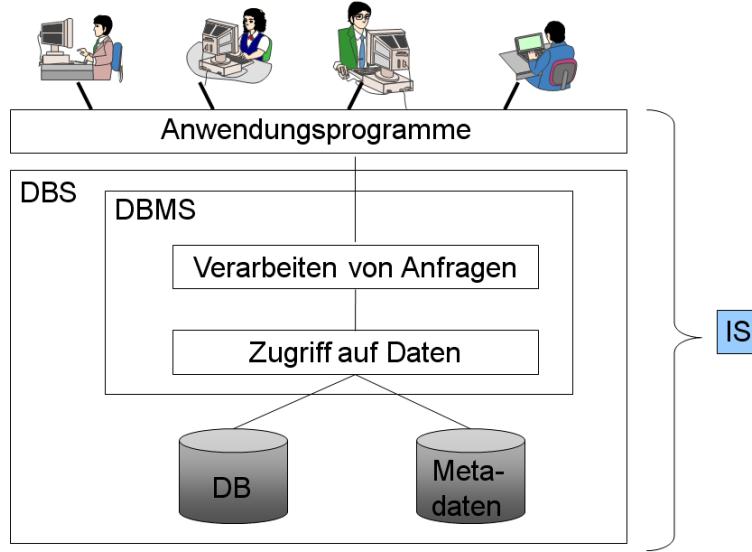


Abbildung 1.1: Komponenten

1.3.1 Aufgaben eines Datenbanksystems

Das DBVS fungiert dabei als Schnittstelle zwischen den Benutzern und einer Datenbank. Es stellt sicher, dass die Benutzer auf ihre Daten in adäquater und effizienter Weise unter zentralisierter Kontrolle zugreifen können und dass die Daten selbst gegen Hard- und Software-Fehler resistent und über längere Zeiträume hinweg vorhanden sind. Das DBVS agiert in der Kommunikation dabei als Server, die Anwendungen als Client. Die vom Server umzusetzenden Aufgaben umfassen dabei:

- ▶ Erzeugen neuer Datenbanken sowie der entsprechenden Datei(en) um Daten persistent zu speichern.¹
- ▶ Interpretation von Anfragen, die von Frontends an die Datenbank gerichtet werden sowie die Rückgabe bzw. Manipulation von Datensätzen, je nach den in der Abfrage-Anweisung enthaltenen Kriterien.
- ▶ Sicherstellung der Datenintegrität, so dass nur geprüfte Daten in die Datenbank aufgenommen werden.
- ▶ Implementierung von Sicherheitsmechanismen, um nicht autorisierten Personen den Zugang zur Datenbank und deren Informationen zu verwehren.
- ▶ Pflege eines Katalogs von Objekten der Datenbank, wie beispielsweise Informationen über den Besitzer der Datenbank sowie die Tabellen, die diese enthält. Kataloge, die Datenbankobjekte beschreiben, werden Metadaten (Daten über Daten) oder bei Datenbanken auch Data Dictionary genannt.
- ▶ Generierung und Pflege einer Protokolldatei, die Informationen über alle Veränderungen an der Datenbank bereit hält. Auf diesem Weg ist es möglich, eine Datenbank mit Hilfe eines zuvor erstellten Backups im Zusammenwirken mit Informationen aus der Protokolldatei vollständig zu rekonstruieren, falls ein Hardwarefehler aufgetreten ist. Solche Protokolldateien werden Transaktionsprotokolle genannt.

¹Im Lateinischen bedeutet das Verb 'persistere' stehen bleiben, verharren. Im Englischen bedeutet 'persistent' u.a. andauernd, beständig. In der Informatik hat die persistente Speicherung von Daten die spezielle Bedeutung, dass die Daten unabhängig von der Programmausführung oder dem Ein-/Ausschalten des Rechners erhalten bleiben.

- ▶ Handhabung konkurrierender Zugriffe, damit viele Benutzer ohne nennenswerte Verzögerung gleichzeitig auf den Datenbestand zugreifen können. Konkurrierende Zugriffe werden durch temporäre Sperren auf einzelne Datensätze einer Tabelle abgewickelt.
- ▶ Ausführen von gespeicherten Prozeduren, die in der Regel vorkomplizierte Abfragen darstellen. Gespeicherte Prozeduren beschleunigen die Ausführung von häufig verwendeten Abfragen, da die Serveranwendung für die Interpretation und gegebenenfalls Optimierung der Abfrage keine zusätzliche Zeit aufwenden muss.
- ▶ Bewahren der Datenkonsistenz sowie Überwachen der Integritätsregeln, um Beschädigungen der Daten in den Tabellen zu verhindern.

Im Gegensatz hierzu sind Frontend-Anwendungen dafür zuständig, Anfragen an das Datenbank-System zu formulieren und dorthin abzusenden sowie die Verarbeitung (Manipulation) der Datensätze vorzunehmen, die das Datenbank-Serversystem zurückgeliert hat. Frontend-Anwendungen handhaben sämtliche Operationen wie Anzeige, Formatierung sowie Ausdruck der Daten.

1.3.2 Datenmodelle

Isoliert betrachtet stellt ein DBVS eine Kontrollinstanz dar, über die Anwendungsprogramme sowie Dialog-Benutzer auf eine große Datensammlung, die Datenbank, zugreifen können. Zur Charakterisierung eines Datenbanksystems sind mehrere Aspekte zu betrachten:

- ▶ Welche Sprachkonzepte stellt ein DBVS seiner Außenwelt zur Verfügung bzw. welche Sicht wird Programmen und Benutzern durch das DBVS auf eine Datenbank gewährt? Wesentlich für die Beantwortung dieser Frage wird die Beschreibung von Datenmodellen sein, die im Zusammenhang mit Datenbanken entwickelt wurden.
- ▶ Welche Übersetzungsorgänge müssen in einem DBVS ablaufen um von der Sprachebene des Benutzers bzw. des Anwendungsprogramms auf die Daten zu kommen?
- ▶ Was muss ein DBVS bereitstellen um gleichzeitig mehreren Benutzern bzw. Anwendungsprogrammen einen adäquaten und zuverlässigen Zugriff auf eine gemeinsame Datenbank zu gewähren.

Die Realisierung dieser Aspekte sind vom Datenmodell abhängig, wobei klassische Datenbanksysteme den Umgang mit strukturierten Daten unterstützen.² Jedes Datenbankverwaltungssystem basiert auf einem Datenmodell, das die Infrastruktur für die Modellierung der realen Welt zur Verfügung stellt. Das Datenmodell legt die Modellierungskonstrukte fest, mittels derer man ein Informationsabbild der realen Welt bzw. des relevanten Ausschnitts generieren kann. Es beinhaltet die Möglichkeit zur:

- ▶ Beschreibung der Datenobjekte und ihrer Beziehungen (auf strukturierte und formale Weise)
- ▶ Festlegung der anwendbaren Operatoren und deren Wirkung
- ▶ Beschreibung der Integritätsbedingungen

Das Datenmodell legt damit die generischen Strukturen und Operatoren fest, die man zur Modellierung einer bestimmten Anwendung einsetzen kann.

²Für eine Verarbeitung unstrukturierter Daten, beispielsweise Texte von E-Mails oder Produktbeschreibungen, sowie semi-strukturierter Daten existieren auch passende Datenbankverwaltungssysteme im Kontext so genannter NoSQL-Datenbanken. Diese sind aber kein Bestandteil dieses Skripts.

1.4 Entwicklungsgeschichte

Die Entwicklungsgeschichte von Datenbanksystemen lässt sich in fünf Generationen gliedern. Die erste Generation an DBS basierte auf einer Datei. Dabei hatte jedes Anwendungsprogramm seine eigene Datei mit eigenem Dateiformat. Lediglich die Ein-/Ausgabe-Routinen des Datenverwaltungssystems wurden durch eine prozedurale Sprache gemeinsam genutzt. Meist handelte es sich bei diesen Dateien um sequentielle Dateien auf Magnetbandgeräten. Die Nachteile dieser Lösungen sind im Abschnitt 1.1 beschrieben.

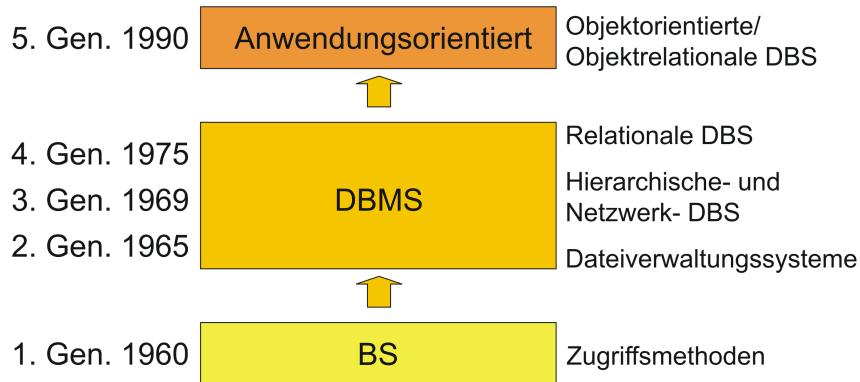


Abbildung 1.2: Geschichte von Datenbanksystemen

Auch die zweite Generation basierte auf einer einzigen Datei. Änderungsdienste, Sortierung, Datenaufbereitung, Listengeneratoren sowie einfache deskriptive Sprachen kamen erweiternd hinzu. Als Zugriffsmethoden standen SAM und ISAM (Index Sequential Access Method) zur Verfügung. Hiermit konnten Systeme erstmals geräteunabhängig entwickelt werden. Die anderen Nachteile der Dateilösung bleiben jedoch erhalten.

Ab der dritten Generation werden alle Daten zentral zusammengefasst. Der Zugriff der Anwendungen wird über die Funktionen des Datenbankverwaltungssystems durchgeführt. Dies hat die Vorteile einer geringeren, kontrollierten Redundanz sowie zentrale Kontrolle aller Zugriffe auf Daten mit einer größeren Widerspruchsfreiheit (Konsistenz) der Daten. Die Benutzerschnittstelle bleibt stabil bei Änderung der Datenorganisation und der Anwendungen, wodurch ein höherer Grad an Datenunabhängigkeit erreicht wird.

Das relationale Datenmodell (Relationenmodell) wurde von Codd im Jahre 1970 eingeführt. Kommerziell zur Verfügung stehende Systeme gab es ab 1980. Das relationale Datenmodell trennt das physische von dem logischen Datenmodell, wodurch ein sehr hoher Grad an Datenunabhängigkeit erreicht wird.

Zahlreiche Weiterentwicklungen zielen darauf ab, die fünfte oder postrelationale Generation von Datenbanksystemen bereitzustellen. Ein zentraler Ansatz ist dabei die Berücksichtigung von Entwicklungen im Bereich der Programmiersprachen, speziell der Objektorientierung. Das Zusammenbringen von Objektorientierung und Datenbanktechniken wird seit Ende der 80er Jahre betrieben und hat im wesentlichen zwei Systemkategorien hervorgebracht: Objektorientierte Datenbanksysteme und Objektrelationale Datenbanksysteme. Objektorientierte Systeme konnten sich jedoch nicht allgemein durchsetzen, sie finden sich nur in Nischenmärkten wie beispielsweise im Konstruktionsbereich, da hier komplexe zusammengesetzte Objekte verarbeitet werden müssen. Die Hersteller relationaler Datenbanken haben jedoch die Grundideen der Objektorientierung in ihre relationalen Produkte aufgenommen, so dass die Vorteile auch auf Basis der relationalen Welt eingesetzt werden können. Man bezeichnet diese Modellerweiterung als als objektrelationales Modell, das in Kapitel 6 betrachtet wird.

Bei Datenmodellen gab es seit den 80er Jahren des 20. Jahrhunderts wenig neue Ansätze. Entwicklungen im Datenbankbereich konzentrieren sich mehr auf die Optimierung des Einsatzes sowie die Unterstützung von Anwendungen und deren Programmierung. So wurde beispielsweise die Verarbeitung von XML-Dokumenten sowie analytische Funktionen standardisiert und in relationale Systeme integriert. Neben allgemein einsetzbaren Datenbanksystemen wurden hierdurch auch anwendungsspezifische Datenbanksysteme entwickelt. Ein wichtiger Grund hierfür ist, dass spezielle Anwendungsbereiche dedizierte Anforderungen an ein DBVS haben. So ist beispielsweise eine Versionsverwaltung für CAD- oder CASE-Datenbanken fundamental, für Banken dagegen nicht relevant. Aufgrund der Spezialisierung sinkt jedoch der Kreis potentieller Kunden, wodurch diese Systeme entweder sehr teuer sind oder im Rahmen der Datenbankforschung entstehen. Beispiele für Anwendungsgebiete spezialisierter DBVS sind:

- ▶ Wissensverarbeitung – Deduktive Datenbanksysteme
- ▶ Temporale Datenbanken
- ▶ Multimediasysteme
- ▶ Geografische Informationssysteme
- ▶ CAD/CASE-Systeme
- ▶ Data Warehousing
- ▶ Naturwissenschaftliche Versuchsanalyse

Neuere Entwicklungen im Datenbankbereich haben das Ziel, große und dabei auch unstrukturierte Daten in hochskalierbaren, verteilten Systemen zu speichern und zu verarbeiten. Solche Lösungen werden unter dem Begriff NoSQL-Datenbanksysteme zusammengefasst. Hierbei gibt es jedoch im Vergleich zu relationalen und objektrelationalen Produkten in der Programmierung keinen übergreifenden Standard. Bei Einsatz ist daher immer eine Einarbeitung in das jeweilige Produkt und die dazugehörige Programmierschnittstelle notwenig, eine Portierung auf ein anderes System ist nur mit hohem Aufwand verbunden.

1.5 Anforderungen an ein Datenbanksystem

Relationale Datenbanksysteme sind heute erfolgreich in viele kommerzielle Anwendungen in Wirtschaft und Verwaltung integriert. Dabei können mehrere Benutzer auf ein und dieselbe Datenbank zeitgleich zugreifen, ohne sich gegenseitig zu beeinflussen oder zu stören. Darüber hinaus kann ein DBVS unautorisierte Datenzugriffe verhindern, zentrale Integritätsbedingungen prüfen und durch geeignete Sicherungsmaßnahmen einen zuverlässigen Betrieb gewährleisten. Zur Behandlung der Aufgaben kennen Mehrbenutzer-DBVS das Konzept der Transaktion, worunter man generell eine Folge von Operationen versteht, welche eine gegebene Datenbank in ununterbrechbarer Weise von einem konsistenten Zustand in einen konsistenten Zustand überführt. Operationen, die einen Geschäftsvorfall einer Anwendung wiederspiegeln, werden deshalb meist zu einer Transaktion zusammengefasst. Für einen sicheren Betrieb ist es daher wesentlich, dass die Verarbeitung von Transaktionen die folgenden vier Eigenschaften sicherstellt, welche zusammen unter dem Akronym ACID-Prinzip bekannt sind:

Atomarität (atomicity): Eine Transaktion (Datenbankbefehle, die zu einem Geschäftsvorgang gehören) wird aus Sicht des Benutzers vollständig oder gar nicht ausgeführt. Wenn ein Programm Änderungen auf der Datenbank durchführt, werden diese nur dann für andere Benutzer sichtbar, falls das Programm vollständig abgearbeitet werden konnte und dabei keine

'Fehler' entdeckt wurden. Wird eine Ausführung vor ihrem eigentlichen Ende abgebrochen, erscheint die Datenbank so, als wäre diese Ausführung nie gestartet worden, durchgeführte Änderungen werden zurückgenommen.

Konsistenz (consistency): Alle Integritätsbedingung der Datenbank werden eingehalten, d.h. eine Transaktion hinterlässt stets einen konsistenten Zustand, falls sie in einem solchen gestartet wurde. So ist es beispielsweise nicht möglich, einen Datensatz einzufügen, der eine Schlüsselbedingung verletzt.

Logischer Einbenutzerbetrieb (isolation): Ein Programm läuft isoliert von anderen Transaktionen ab. Dem Programm werden nur 'gesicherte' Daten aus der Datenbank zur Verarbeitung zur Verfügung gestellt, welche Teil eines konsistenten Zustands sind.

Dauerhaftigkeit (durability): Falls eine Transaktion dem Benutzer als erfolgreich beendet gemeldet wird, überleben alle in der Transaktion durchgeführten Änderungen (fast) jeden danach auftretenden Hard- oder Softwarefehler.

Neben diesen Anforderungen weisen die Standardanwendungsbereiche eine Reihe gemeinsamer Kennzeichen auf, die von relationalen Datenbanksystemen sehr gut unterstützt werden können:

- ▶ Die Daten sind einfach strukturiert.
- ▶ Einfache Datentypen sind meist ausreichend.
- ▶ Die meisten Vorgänge sind kurz und wiederkehrend (Überweisung, Einzahlung, Abhebung, Kontostandsabfrage).
- ▶ Es werden hohe Transaktionsraten benötigt, Endbenutzer erwarten eine unmittelbare Bearbeitung ohne nennenswerte Verzögerungen.
- ▶ Änderungen sind In-Place durchzuführen, der alte Wert wird überschrieben.

Kapitel 2

Datenbankentwurf mit dem Entity-Relationship-Modell

Datenbankentwurf ist der Prozess der Bestimmung der Organisation und des inhaltlichen Aufbaus einer Datenbank. Dieser Prozess ist durchzuführen, wenn für eine bestimmte, wohlumrissene Anwendung eine Datenbank eingerichtet werden soll, die für die Anwendung Daten in geeigneter Weise aufnehmen kann. Eine kurze Definition des Begriffs Datenbankentwurf ist nach Vossen:

Definition 2.1 *Die Aufgabe des Datenbankentwurfs ist der Entwurf der logischen und physischen Struktur einer Datenbank in der Art, dass die Informationsbedürfnisse der Benutzer in einer Organisation für bestimmte Anwendungen adäquat befriedigt werden können.*

Für diese nicht-triviale Aufgabe werden spezielle Techniken benötigt, wobei das Problem in eine Reihe von Einzelschritten zerlegt wird. Das Entity-Relationship-Modell dient dabei der zentralen Phase im Entwurfsprozess, dem konzeptionellen Entwurf.

2.1 Phasen des Entwurfsprozesses

Abbildung 2.1 zeigt eine vereinfachte, typische Phasen-Aufteilung des Lebenszykluses eines Informationssystems, wobei im folgenden die Phasen des Datenbankentwurfs kurz vorgestellt werden. Hierbei werden insbesondere datenbanktypische Aspekte betrachtet, für eine ausführliche Betrachtung des Entwurfs von Anwendungsprogrammen sei hier auf die zahlreiche Literatur sowie andere Vorlesungen verwiesen.

2.1.1 Anforderungsanalyse und -spezifikation

Im Rahmen der Anforderungsanalyse werden neben den Anforderungen der Anwendungsebene auch die Anforderungen aller potentiellen Benutzer an die neu einzurichtende Datenbank erhoben. Diese Anforderungen sind zunächst zu sammeln und zu dokumentieren und werden anschließend einer Analyse unterzogen. Wesentlich ist im Bereich der Datenbanksystem-Anforderungen, dass man einerseits feststellt, über *was* Daten gespeichert werden sollen, und andererseits, *wie* diese Daten bearbeitet werden sollen. Daher unterscheidet man zwischen zwei Arten von Anforderungen:

Informationsanforderungen: Hierunter fallen alle statischen Informationen, die das Datenbanksystem benutzen wird. Diese umfassen beispielsweise Angaben über Daten, Realwelt-Objekte und deren Typen, Objekteigenschaften und deren Wertebereiche, Beziehungen und

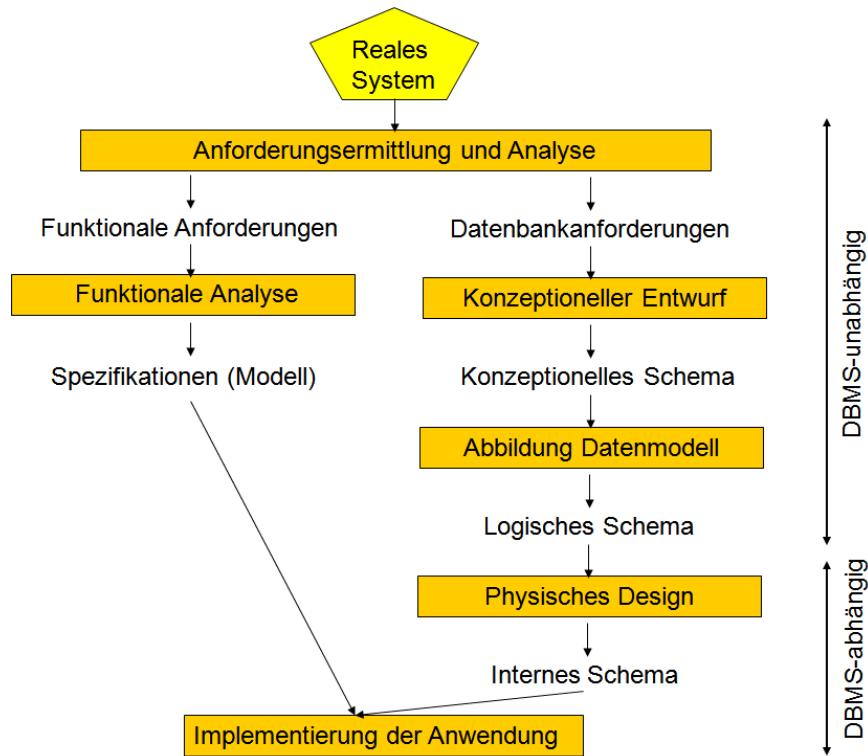


Abbildung 2.1: Entwicklungsphasen eines Informationssystems

Abhängigkeiten zwischen Daten und Objekten, Identifikatoren (Schlüssel) für Objekte sowie Integritätsbedingungen, über die die Konsistenz einer Datenbank definiert wird. Bei der Erhebung von Informationsanforderungen geht es also um die Syntax und die Semantik der in der Datenbank abzulegenden Daten.

Bearbeitungsanforderungen: Hierzu rechnet man die Spezifikation von dynamischen Aktivitäten und Prozessen, welche später auf der Datenbank ablaufen sollen. Dazu zählen Informationen darüber, welche Prozesse (Abfragen, Auswertungen, Updates, usw.) auf der Datenbank ausgeführt werden. Daneben sind Informationen darüber zu sammeln, wie häufig die einzelnen Prozesse durchgeführt werden sollen, ob zwischen einzelnen Prozessen gewisse Reihenfolgen einzuhalten sind, welches Datenvolumen und welche Datenqualität von den einzelnen Prozessen benötigt wird, usw. Zu diesen Anforderungen gehören insbesondere auch Aussagen über die Verfügbarkeit oder die Sicherheit der Daten in der Datenbank oder die Spezifikation von Zugriffsrechten.

Typischerweise werden während dieser Phase des Entwurfsprozesses folgende Aktivitäten durchgeführt:

- ▶ Identifikation der wesentlichen Benutzergruppen und Anwendungsbereiche der zu entwerfenden Datenbank
- ▶ Sichtung existierender Dokumentation
- ▶ Fragebögen und Interviews

2.1.2 Konzeptioneller Entwurf

In der zweiten Phase, dem DBVS-unabhängigen konzeptionellen Entwurf, besteht die Aufgabe des Designers grundsätzlich darin, eine konzeptionelle Globalsicht der Anwendung zu erstellen, wobei rein die fachliche Sicht auf die Anforderungen berücksichtigt werden. Eingabe für diese Phase ist die Anforderungsspezifikation, Ausgabe ein konzeptionelles, zielsystem-unabhängiges Datenbankschema. Während dieser Phase wird die Anforderungsspezifikation modelliert und formalisiert. Dazu gehören:

- ▶ Festlegung der Objekttypen über die Daten gespeichert werden
- ▶ Festlegung der zu speichernden Merkmale der Objekttypen einschließlich der Identifizierungsmerkmale
- ▶ Definition der Beziehungen zwischen Objekttypen
- ▶ Definition der Integritätsbedingungen

2.1.3 Logischer Entwurf

In der dritten Phase, dem logischen Entwurf, erfolgt eine Übersetzung oder Transformation des konzeptionellen Modells in das Datenmodell des zu verwendeten Datenbanksystems. Wesentliche Hilfsmittel zur Durchführung dieser Phase sind Transformationsregeln, die angeben, wie die Konstrukte des konzeptionellen Schemas in die des Zieldatenmodells zu übersetzen sind. Im Kapitel zum relationalen Datenbankmodell wird die Überführung von Entity-Relationship-Modellen in das relationale Datenbankmodell vorgestellt.

Ein logischer Entwurf kann auch in mehreren Teilschritten durchgeführt werden. So kann man nach der Transformation das Resultat noch optimieren. Im Fall des relationalen Modells sind unter allgemeinen Kriterien die Normalisierungsschritte zu verstehen, die in Kapitel 4 beschrieben werden.

2.1.4 Physischer Entwurf

Der physische Entwurf besteht in einer Definition der DBVS-internen Datenrepräsentation sowie der damit zusammenhängenden Systemparameter. Unter Berücksichtigung der zuvor erhobenen Bearbeitungsanforderungen sind geeignete Speicherungsstrukturen für die einzelnen Elemente des konzeptionellen Schemas sowie Zugriffsmechanismen hierauf festzulegen. Wichtig für das Laufzeitverhalten eines Datenbanksystems ist vor allem eine Minimierung der Zugriffszeiten auf den Sekundärspeicher. Dazu müssen die Datenstrukturen einen effizienten Zugriff auf die relevanten Daten durch effiziente Zugriffspfade erlauben.

Der physische Entwurf ist systemspezifisch durchzuführen, hier unterscheiden sich die verschiedenen DBVS in ihren Möglichkeiten erheblich. So sollte jeder Entwickler beispielsweise die Optionen des später genauer betrachteten Befehls CREATE TABLE beim eingesetzten DBVS betrachten, typischerweise finden sich hier einige Optionen, die für den physischen Entwurf wichtig sind.

2.2 Allgemeine Abstraktionskonzepte

Ein Datenmodell dient allgemein der Herstellung einer Abstraktion zum Zweck der Beschreibung eines Problems. Unter Abstraktion versteht man dabei einen mentalen Prozess, der dazu dient,

einige Charakteristika und Eigenschaften einer Menge von Objekten zu deren Beschreibung auszuwählen und gleichzeitig andere, nicht relevante auszuschließen. Wesentlich für Datenbanken bzw. für den Entwurf einer Datenbank sind die folgenden drei Abstraktionsmechanismen, die sich in jedem konzeptionellen Datenmodell identifizieren lassen:

Klassifikation: Eine Klassifikation dient der Definition von Klassen mit Elementen bzw. Objekten mit gemeinsamen Eigenschaften. Eigenschaften werden in diesem Zusammenhang als Attribute bezeichnet. Mathematisch geht es hierbei also lediglich um Mengenbildung.

Generalisierung bzw. Spezialisierung: Eine Verallgemeinerung oder Generalisierung definiert eine Teilmengenbeziehung zwischen den Elementen verschiedener Klassen. So ist beispielsweise die Klasse Fahrzeug eine Verallgemeinerung der Klasse Fahrrad, denn jedes Fahrrad ist ein Fahrzeug. Umgekehrt lässt sich die Klasse Fahrrad als Spezialisierung von Fahrzeug auffassen. Wesentlich bei diesem Abstraktionskonzept ist die Vererbungseigenschaft. Alle Attribute, die für die allgemeinere Klasse definiert sind, werden an jede Teilmenge vererbt, d.h. sie sind auch für deren Elemente implizit definiert. Hat beispielsweise die Klasse Fahrzeug die Attribute Baujahr und Höhe, so erhält die Spezialisierung Fahrrad automatisch auch diese Attribute und kann dabei noch zusätzliche Attribute wie Rahmenhöhe definieren. Vererbung steht jedoch nicht in jedem Datenmodell explizit zur Verfügung.

Aggregation (Zusammensetzung): Eine Aggregation definiert eine neue Klasse aus anderen, bereits existierenden, die dann Komponenten darstellen. Diese Form der Abstraktion wird beispielsweise dann angewandt, wenn man eine Klasse Auto ausgehend von bereits gegebenen Definitionen der Klassen Motor, Karosserie usw. definiert.

2.3 Das Entity-Relationship-Modell

Das Entity-Relationship-Modell (kurz ER-Modell) wurde 1976 von Peter Chen vorgeschlagen. Es hat heute eine hohe Bedeutung sowohl im Datenbankentwurf als auch in anderen Bereichen der Informatik, in denen die Modellierung von Realwelt-Zusammenhängen auf einer abstrakten Ebene eine wichtige Rolle spielt. In diesem Abschnitt werden die Grundzüge des ER-Modells im Hinblick auf relationale Datenbanken eingeführt. Damit wird einerseits eine Beschränkung vorgenommen, da viele der seit der Vorstellung des ER-Modells vorgeschlagenen Erweiterungen nicht diskutiert werden. Andererseits sind die hier beschriebenen ER-Konzepte ausreichend für den Entwurf der statischen Aspekte von Datenbanken.

Das ER-Modell hat sich aus verschiedenen Gründen als wichtiges Instrument im Datenbankentwurf durchgesetzt:¹

- ▶ Es ist unabhängig von einem bestimmten Datenbanksystem. Deshalb unterliegt es keinen Beschränkungen, die durch eine Systemimplementierung einem Datenmodell möglicherweise aufgezwungen werden.
- ▶ Die Grundkonzepte 'Entity' und 'Relationship' werden einerseits als sehr natürliche Ausdrucksmittel empfunden, andererseits für viele Anwendungen als ausreichend betrachtet.
- ▶ Es unterstützt die Abstraktionskonzepte Klassifikation, Aggregation und Generalisierung.

¹Auf dem Markt sind eine Reihe von leistungsfähigen Entity-Relationship-Werkzeugen erhältlich, die den Entwurf und die Dokumentation von Datenmodellen unterstützen und dabei die so erstellten Modelle auch direkt in die relationale Datenbankwelt überführen können. Zu den bedeutendsten Produkten in diesem Bereich gehören Erwin von Computer Associates, Powerdesigner Data Architect von Sybase und ER/Studio vom Embarcadero, die alle im Bereich zwischen 3000 und 4000 Dollar liegen.

Mit Hilfe eines ER-Modells können so komplexe Sachverhalte vereinfachend und grafisch anschaulich dargestellt werden, Mitarbeiter aus verschiedenen Bereichen können diese Modelle schnell verstehen. Systementwickler können sich aufgrund der Abstraktion auf das Wesentliche konzentrieren ohne gleichzeitig den Überblick auf das gesamte System zu verlieren. Auch werden aufgrund der klaren Semantik Misverständnisse verhindert.

2.3.1 Entities und Attribute

Wohlunterscheidbare Dinge der realen Welt werden Entities (zu deutsch: Entitäten, Dinge, Objekte, Gegenstände) genannt. Beispiele hierfür sind die Stadt München, die Person Herr Müller, das Auto des Autors, der Student in der ersten Reihe links. Einzelne Entities, die ähnlich, vergleichbar oder zusammengehörig sind, werden zu Entity-Mengen zusammengefasst. Beispiele hierfür sind Städte, Personen, Fahrzeuge, Studenten der Wirtschaftsinformatik. Entities dürfen in mehreren Mengen enthalten sein, die Mengen müssen damit nicht disjunkt sein. Im Folgenden werden Entity-Mengen stets mit Groß-, einzelne Entities mit Kleinbuchstaben bezeichnet.

Entities besitzen Eigenschaften (wie Farbe, Größe), deren konkrete Ausprägungen als Werte bezeichnet werden. Die Zusammenfassung aller möglichen bzw. zugelassenen Werte für eine Eigenschaft ist der Wertebereich (Domain). Eigenschaften einer Entity-Menge werden Attribute genannt. Für die Modellierung mancher Realitätsausschnitte kann für bestimmte Attribute kein Wert angegeben werden. So kann es sein, dass Angestellte kein Festnetztelefonanschluss besitzen oder dass sie ihre Festnetz-Nummer – etwa aus persönlichen Gründen – nicht bekannt geben. Da ein Attributwert immer ein Wert des zugehörigen Wertebereichs sein muss, sind bei einfachen Wertebereichen solche Situationen nicht zulässig. Um Situationen dieser Art darstellen zu können, bedient man sich so genannter Nullwerte. Dabei ist es meist ausreichend, den Wertebereich eines Attributs um den Wert NULL zu erweitern und diesen wie einen realen Wert zu behandeln. Der Attributwert kann damit NULL² sein, wenn für das Entity bezüglich dieses Attributs kein Wert angegeben werden kann oder falls der Wert unbekannt ist.

ER-Modelle werden in Form von Grafiken dargestellt. Das Symbol für einen Entity-Typ ist ein Rechteck, in dem der Name des Typs (meist im Singular) steht. Die dazugehörigen Attribute werden als mit diesem Rechteck verbundene Ovale dargestellt. Als erstes Beispiel zeigt Abbildung 2.2 eine Darstellung einer Buch-Menge.

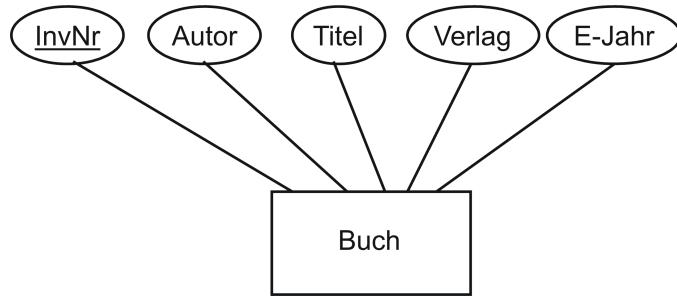


Abbildung 2.2: Entity-Diagramm für Bücher

Das Unterstreichen des Attributs InvNr zeigt an, dass dieses Attribut eine Identifikationsfunktion besitzt. Zur Beschreibung eines speziellen Entities ist häufig nicht die Angabe aller Attributwerte erforderlich, statt dessen kann es ausreichen, nur für gewisse Attribute den Wert anzugeben, durch welches das Entity dann eindeutig identifiziert wird. Dies ist in einer Bibliothek die Inventarnummer des Buchs. Solche Attribute werden als Schlüsselattribute und ihre Zusammenfassung als Primärschlüssel für die Entity-Menge bezeichnet. Damit über den Primärschlüssel auch immer

²Häufig wird auch die Symbole '?' und '-' für NULL-Werte verwendet.

eine Identifizierung eines Entities möglich ist, dürfen Primärschlüsselattribute nie den Wert NULL annehmen.

Da alle Entities einer Menge wohlunterscheidbar sind, ist die Menge aller Attribute stets ein möglicher Primärschlüssel. Meist reichen jedoch weniger Attribute, häufig auch nur ein Attribut aus. Es ist daneben auch denkbar, dass sich für eine Entity-Menge mehr als ein Primärschlüssel angeben lässt. Die möglichen Alternativen sind die Schlüsselkandidaten. Für eine Entity-Menge 'Studiengangsleiter' sind Personalnummer, Telefonnummer und Raumnummer mögliche Schlüsselkandidaten. In einem solchen Fall wird ein Schlüssel ausgewählt.

Definition 2.2 Sei $A = \{A_1, \dots, A_m\}$ Menge der Attribute der Entity-Menge E .
 $K \subseteq A$ heißt *Schlüsselkandidat* von E gdw.
 $\forall e_i, e_j \in E$ mit $e_i \neq e_j$ gilt $K(e_i) \neq K(e_j)$.

Der durch unterstreichen gekennzeichnete Schlüsselkandidat ist der Primärschlüssel der Entity-Menge.

Bei einer genaueren Betrachtung des Beispiels in Abbildung 2.2 fällt auf, dass die Modellierung die Realität von Büchern nur unzureichend erfasst, denn jedes Attribut wird dort als einwertig angenommen. Ein Buch kann jedoch mehrere Autoren haben und die Verlagsinformation ist häufig aus einem Verlagsnamen und einem Verlagsort zusammengesetzt.

Mehrwertige Attribute werden in ER-Diagrammen durch Doppelovale gekennzeichnet, für zusammengesetzte Attribute werden deren Komponenten ebenfalls durch Ovale dargestellt und über Kanten mit der Zusammensetzung verbunden. Als Beispiel zeigt Abbildung 2.3 eine erweiterte Darstellung der Buchinformation.

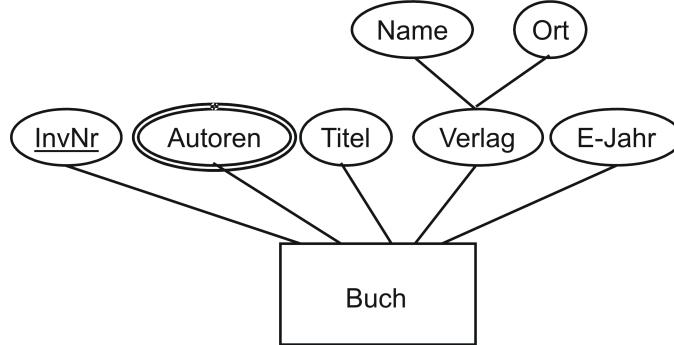


Abbildung 2.3: Erweitertes Entity-Diagramm für Bücher

Es sei bemerkt, dass Mehrwertigkeit und Zusammensetzung nicht auf eine einmalige Anwendung beschränkt ist. So kann es mehrwertige Attribute geben, die aus mehreren anderen zusammengesetzt sind. Auch kann es zusammengesetzte Attribute geben, deren Komponenten wiederum einwertig, mehrwertig oder zusammengesetzt sind. Da Diagramme dabei schnell unübersichtlich werden, wird dies in den Beispielen nicht verwendet.

Aus den obigen Erläuterungen ergibt sich folgende Definition von Entity-Typen:

Definition 2.3 Ein Entity-Typ hat die Form ' $E = (X)$ '. Er besteht aus einem Namen E , einem Format X und einem Primärschlüssel K , welcher aus Elementen von X zusammengesetzt ist. Die Elemente des Formats X werden dabei wie folgt notiert:

- Einwertige Attribute: A

- ▶ *Mehrwertige Attribute:* $\{A\}$
- ▶ *Zusammengesetzte Attribute:* $A(B_1, \dots, B_k)$

Schlüsselattribute werden dabei unterstrichen.

Der in Abbildung 2.3 dargestellte Entity-Typ lässt sich damit wie folgt beschreiben:

$$\text{Buch} = (\underline{\text{InvNr}}, \{\text{Autor}\}, \text{Titel}, \text{Verlag}(\text{Name}, \text{Ort}), \text{E-Jahr})$$

Man darf bei dieser Darstellung nicht vergessen, dass Attribute immer einen zugeordneten Wertebereich haben. Falls jedoch aus dem Zusammenhang klar ist, welcher Wertebereich gemeint ist, wird nur der Name angegeben. Die vollständige Beschreibung des Buchbeispiels sieht so aus:

$$\text{Buch} = (\underline{\text{InvNr}}:\text{int}(15), \{\text{Autor}:\text{char}(30)\}, \text{Titel}:\text{char}(30), \\ \text{Verlag}(\text{Name}:\text{char}(30), \text{Ort}:\text{char}(30)), \text{E-Jahr}:\text{int}(4))$$

Neben der bisher verwendeten Originalnotation von Cheng innerhalb der Diagramme finden sich in der Literatur diverse kompaktere Darstellungsformen für Entity-Typen, die sich teilweise an UML anlehnern. Einige Beispiel hierfür zeigt Abbildung 2.4.

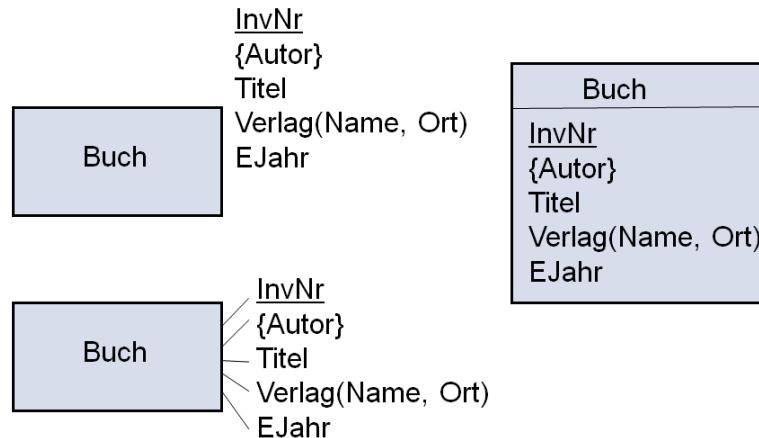


Abbildung 2.4: Darstellungsalternativen

Definition 2.4 Eine Entity-Menge zu einem Entity-Typ $E = (X)$ ist eine Menge von Entities, die dem Aufbau des Typs entsprechen. Dies wird auch als Inhalt des Entity-Typs bezeichnet.

2.3.2 Relationships

Verschiedene Entity-Mengen einer vorgegebenen Anwendung können miteinander in Beziehung stehen. So werden beispielsweise Bücher von Lesern *entliehen*, so dass durch den Entleihvorgang ein bestimmtes Buch mit einem bestimmten Leser in Beziehung steht. An einer Beziehung (Relationship) sind folglich immer Entities beteiligt. Gleichartige Beziehungen werden zu einer Relationship-Menge mit einem Namen zusammengefasst. Diese könnte in diesem Beispiel *Ausleihe* genannt werden.

Auch Beziehungen können eigene Attribute besitzen, welche spezielle Eigenschaften zum Ausdruck bringen, die erst durch das Herstellen der Beziehung relevant werden und für die einzelnen Entity-Mengen bedeutungslos sind. Ein Beispiel hierfür ist das Rückgabedatum der Beziehung Ausleihe. Auch hier können die Attribute zusammengesetzt und mehrwertig sein.

Innerhalb der grafischen Darstellungen werden Relationships durch Rauten dargestellt, in denen der Name der Beziehung eingetragen wird. Diese wird durch Kanten mit den beteiligten Entity-Typen verbunden. Die Kanten sind dabei ungerichtet. Attribute werden analog zu Entity-Typen durch Ovale dargestellt. Abbildung 2.5 zeige das Beispiel einer Klausur-Beziehung zwischen Studenten und Professoren mit den Attributen Datum und Note.

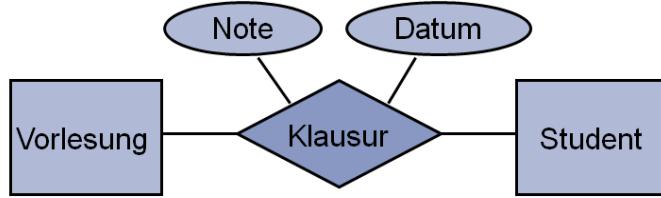


Abbildung 2.5: Klausur-Beziehung

Definition 2.5 Eine Relationship-Deklaration hat die Form $R = (Ent, Y)$. Dabei ist R der Name der Beziehung, Ent bezeichnet die Folge der Namen der Entity-Typen, zwischen denen eine Beziehung definiert werden soll, und Y ist eine (möglicherweise leere) Folge von Attributnamen der Beziehung.

An einer Relationship-Deklaration kann ein Entity-Typ auch mehrfach teilnehmen, wobei dies als rekursive Beziehung bezeichnet wird. So kann beispielsweise eine Vorgesetztenbeziehung zwischen Person und Person auftreten. Gleiches gilt für 'verheiratet' oder 'Vater-Sohn'. Zur Verbesserung der Verständlichkeit solcher Beziehungen können Rollennamen eingeführt werden, durch die die Bedeutung des jeweiligen Entity-Typs definiert wird. Rollennamen sind technisch nicht erforderlich, bei rekursiven Beziehungen liefern sie jedoch Attributnamen, die für eine Umsetzung in das relationale Datenmodell benötigt werden. Primär dienen Rollennamen aber der Verbesserung der Verständlichkeit von Modellen, sie können daher auch bei nicht rekursiven Beziehungen verwendet werden.

Beispiel:

$\Rightarrow \text{Vorgesetzter} = ((\text{Chef: Person}, \text{Mitarbeiter:Person}), \{\})$

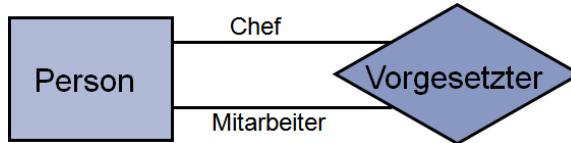


Abbildung 2.6: Rollennamen in Diagrammen

Ist $R = (Ent, Y)$ eine Relationship-Deklaration mit $Ent = (E_1, \dots, E_k)$, dann ist k die Stelligkeit von R . Der Fall $k = 2$ ist in praktischen Anwendungen mit Abstand der häufigste Fall. Ein Beispiel für eine Beziehung höherer Stelligkeit ist in Abbildung 2.7 dargestellt, wobei dieses Beispiel auch zeigt, dass eine dreistellige Beziehung nicht durch drei zweistellige Beziehungen ersetzt werden kann. So enthält die dreistellige Beziehung genau die Lieferungen von Lieferanten an Teilen für ein Projekt, während die untere Darstellung viel allgemeiner ist. Hier sind auch mögliche Lieferteile von Lieferanten enthalten. Zudem sind Lieferungen von Dienstleistungen möglich. Dagegen ist im

unteren Modell nicht erkennbar, welcher Lieferant welche Teile für welches Projekt liefert. Auch wenn Teile für ein Projekt benötigt werden, kann daraus nicht gefolgert werden, dass ein Lieferant, der dieses Teil liefert, dieses Teil auch für das Projekt liefert.

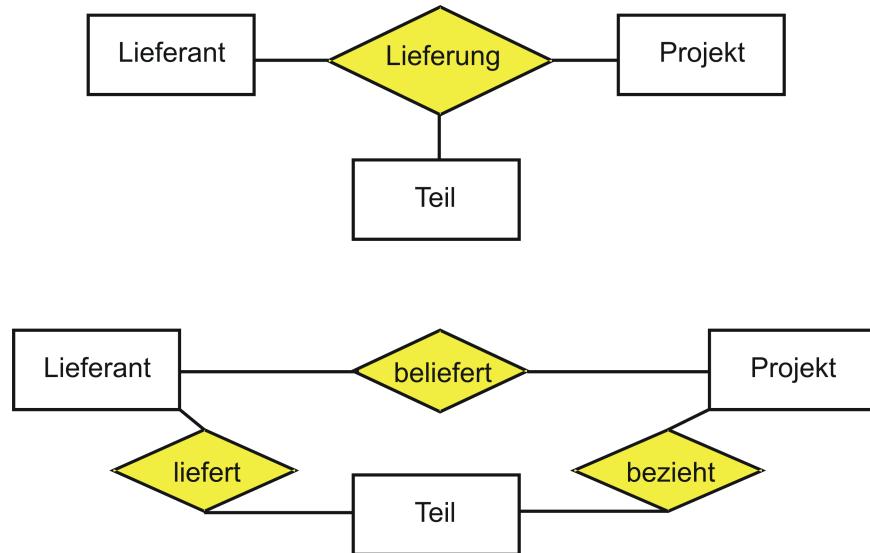


Abbildung 2.7: Eine 3-stellige Beziehung

2.3.2.1 Beziehungstypen für binäre Beziehungen

Den Beziehungstyp, auch Kardinaltätstyp genannt, einer binären Relationship kann man über die Funktionalität der Beziehung charakterisieren. Ein binärer Beziehungstyp R zwischen den Entity-Typen E_1 und E_2 heißt

1:1-Beziehung, falls jedem Entity e_1 aus E_1 höchstens ein Entity e_2 aus E_2 zugeordnet ist und umgekehrt jedem Entity e_2 aus E_2 maximal ein Entity e_1 aus E_1 zugeordnet ist. Man beachte, dass es auch Entities aus E_1 (bzw. E_2) geben kann, denen kein Partner aus E_2 (bzw. E_1) zugeordnet ist. Beispiele sind Abteilungsleitung zwischen Abteilung und Person sowie verheiratet zwischen Personen³. In der Grafik wird hierbei an jede Kante eine 1 gestellt (vgl. Abbildung 2.8).

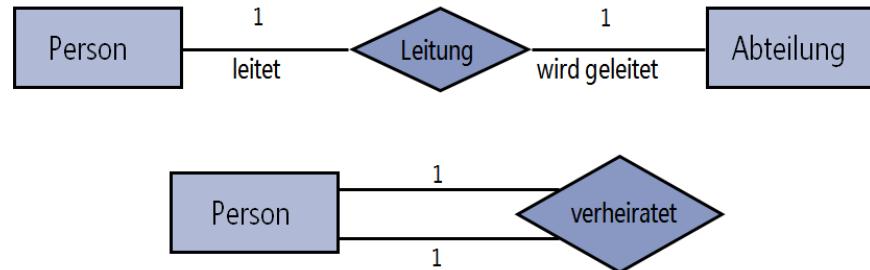


Abbildung 2.8: 1:1-Beziehungen

³zumindest nach europäischem Recht

1:n-Beziehung, falls jedem Entity e_1 aus E_1 beliebig viele (mehrere oder auch gar keins) Entities aus E_2 zugeordnet sein können, aber jedes Entity e_2 aus E_2 mit maximal einem Entity e_1 aus E_1 in Beziehung stehen kann. Beispiele sind 'angestellt_beij' zwischen Unternehmen und Person sowie Personalvorgesetzter zwischen Personen. In der Grafik wird hierbei an die Kante von der Beziehung zum Entity-Typ, von dem mehrere Partner zugeordnet werden können, ein n gestellt, an die andere Kante eine 1 (vgl. Abbildung 2.9).

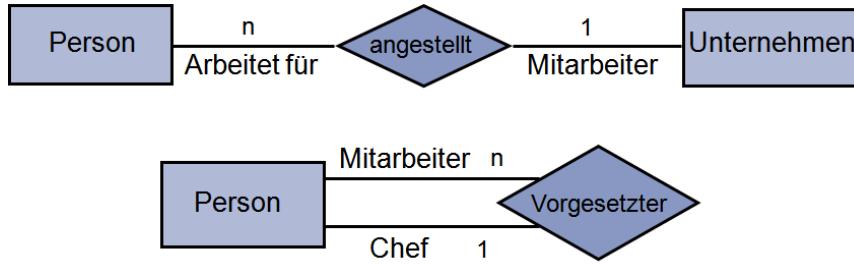


Abbildung 2.9: 1:n-Beziehungen

n:1-Beziehung analog zu 1:n

n:m-Beziehung, falls keinerlei Restriktionen gelten müssen. Jedes Entity aus E_1 kann mit beliebig vielen Entities aus E_2 in Beziehung stehen und umgekehrt darf jedes Entity aus E_2 mit beliebig vielen Entities aus E_1 assoziiert werden. Beispiele sind Mitarbeit zwischen Projekt und Person sowie Bestandteil zwischen Teilen. In der Grafik werden n und m an die Kanten gestellt (vgl. Abbildung 2.10).

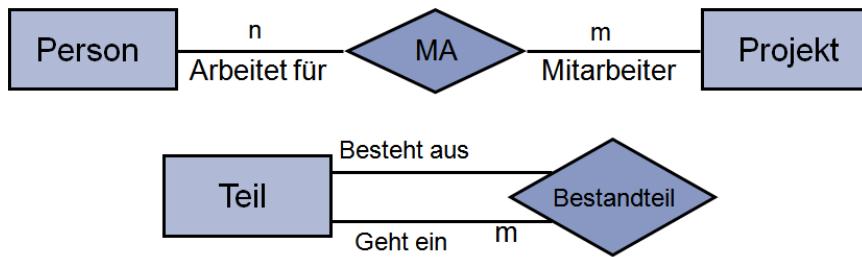


Abbildung 2.10: n:m-Beziehungen

Man beachte, dass die Funktionalitäten Integritätsbedingungen darstellen, die in der zu modellierenden Welt immer gelten müssen. D.h. diese Bedingungen sollen nicht nur im derzeit existierenden Zustand der Miniwelt (rein zufällig) gelten, sondern sie sollten Gesetzmäßigkeiten darstellen, deren Einhaltung erzwungen wird.

Die Angabe von Funktionalitäten wurde bisher nur für binäre Beziehungstypen definiert, sie kann aber auch auf n-stellige Beziehungen erweitert werden. Hiefür sei auf das Buch *Datenbanksysteme* von A. Kemper verwiesen. Wir verzichten bei $n > 2$ auf die Angabe des Beziehungstyps.

Bei obigen Überlegungen wurde zur Feststellung des Beziehungstyps immer geprüft, wie viele Partner man maximal über die Beziehung findet, wobei nur die beiden Fälle 1 und n unterschieden werden. Eine Erweiterung der Notation erlaubt auch die Berücksichtigung, ob ein Entity auch mindestens einmal in die Beziehung eingehen muss. In dieser Schreibweise werden dann nicht nur die beiden Fälle 1 und n eingesetzt, sondern insgesamt die folgenden vier Varianten:

1: genau ein

c: kein oder ein

n: ein oder mehrere

nc: kein, ein oder mehrere

Noch genauere Informationen zu Häufigkeiten der Beziehungsteilnahme liefern die nachfolgende (min,max)-Notation.

2.3.2.2 Die (min,max)-Notation

Beziehungen können präziser als über die Funktionalität der Beziehungstypen definiert werden. Bei der Angabe der Funktionalität ist für ein Entity nur die maximale Anzahl an Beziehungsinstanzen relevant. Wann immer diese Zahl größer als eins ist, wird sie, ohne genauere Aussagen zu machen, mit n oder m (für viele) angegeben. Bei der (min-max)-Notation (auch Kardinalitätsrestriktion genannt) werden nicht nur die Extremwerte eins oder viele angegeben sondern, wann immer möglich, präzise Unter- und Obergrenzen festgelegt. Es wird also auch die minimale Anzahl angegeben, da dies für viele Beziehungstypen eine sinnvolle und manchmal auch essentielle Integritätsbedingung darstellt.

Bei der (min-max)-Notation wird für jeden an einer Beziehung R beteiligten Entity-Typ E_i ein Paar von Zahlen, nämlich min und max ($=\text{kard}(R,E_i)$), angegeben (vgl. Abbildung 2.11). Dieses Zahlenpaar sagt aus, dass jedes Entity dieses Typs mindestens min-mal und höchstens max-mal in der Beziehung steht.

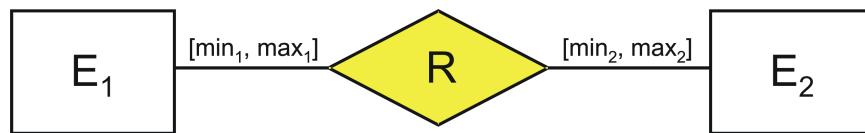


Abbildung 2.11: Angabe von Kardinalitätsrestriktionen

Sonderfälle werden wie folgt gehandhabt:

- ▶ Wenn es Entities geben darf, die nicht an der Beziehung teilnehmen, so wird min mit 0 angegeben.
- ▶ Wenn ein Entity beliebig oft an der Beziehung teilnehmen kann, so wird die max-Angabe mit einem '*' dargestellt.

Übung 2.1 Füllen Sie die Tabelle in Abbildung 2.12 mit Beziehungstypen und Kardinalitäten aus.

2.3.2.3 Schwache Entity-Typen

Bislang wurde immer davon ausgegangen, dass Entities autonom existieren und innerhalb ihrer Entitymenge über die Schlüsselattribute eindeutig identifizierbar sind. In der Realität gibt es jedoch häufig so genannte schwache Entities, bei denen dies nicht gilt. Diese Entities sind:

- ▶ in ihrer Existenz von einem anderen, übergeordneten Entity abhängig und
- ▶ nur in Kombination mit dem Schlüssel des übergeordneten Entities eindeutig identifizierbar.

R	E ₁	E ₂	Beziehungs-typ	kard(R, E ₁)	kard(R, E ₂)
Abteilungsleitung	ABT	Person			
Projektmitarbeit	Person	Projekt			
Parteimitglied	Partei	Person			
Verheiratet	Person	Person			
Vorlesungsteilnahme	Vorlesung	Student			
Belegung	Person	Zimmer			
Eltern	Paar	Kind			

Abbildung 2.12: Übung zu Kardinalitätsrestriktionen

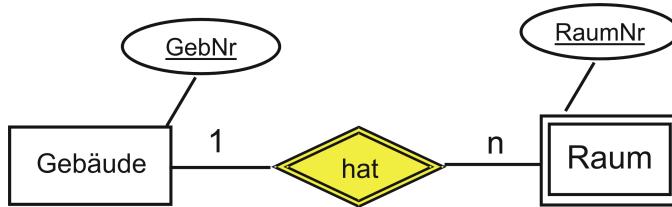


Abbildung 2.13: Schwacher Entitotyp Raum

Man betrachte hierzu das Beispiel in Abbildung 2.13. Die Entities des Typs Raum sind existenz-abhängig von dem Gebäude, in dem der betreffende Raum liegt. Dies ist auch intuitiv einsichtig: Wenn man das Gebäude abreißt, verschwinden damit auch alle in dem betreffenden Gebäude liegenden Räume. Bzgl. eines Primärschlüssels hat der Raum keinen eigenen Schlüsselkandidaten, die Raumnummer wird erst über den Bezug zum Gebäude eindeutig. Daher ist Raum ein schwacher Entity-Typ.

Schwache Entity-Typen werden im Modell durch doppelt umrandete Rechtecke repräsentiert. Die Beziehung zu dem übergeordneten Entity-Typ wird ebenfalls durch eine Verdopplung der Raute dargestellt. Die Beziehung zum übergeordneten Entity-Typ hat meist eine 1:n-Funktionalität oder, in seltenen Fällen, eine 1:1-Funktionalität. Eine n:m-Funktionalität ist nicht möglich, da das übergeordnete Entity, von dem ein schwaches Entity existenzabhängig ist, eindeutig identifizierbar sein muss. Aus diesem Grund hat der schwache Entity-Typ eine min-Wert der Kardinalitätsrestriktion von 1, jedes Entity benötigt für die Existenz ein übergeordnetes Entity.

Obwohl schwache Entity-Typen keinen eigenständigen Primärschlüssel haben, der alle Entities der Entitymenge eindeutig identifiziert, gibt es ein Attribut (oder eine Menge von Attributen), deren Wert alle schwachen Entities, die einem(!) übergeordneten Entity zugeordnet sind, voneinander unterscheidet. In der grafischen Notation werden auch diese Attribute unterstrichen, das Unterstreichen ist aber durch die Existenzabhängigkeit entsprechend zu interpretieren. In dem Beispiel handelt es sich hierbei um das Attribut RaumNr, da alle Räume in demselben Gebäude eine eindeutige Raumnummer haben. Räume werden erst global eindeutig, wenn der Schlüsselwert des übergeordneten Entities, die Gebäudenummer, bekannt ist.

2.3.2.4 Zusammenfassung

Eine vollständige Relationship-Deklaration umfasst damit:

- ▶ eindeutiger Name R
- ▶ Grad der Beziehung n (Anzahl an Entities)
- ▶ Beteiligte Entity-Typen E_1, \dots, E_n
- ▶ Evtl. Festlegung der Rollennamen
- ▶ Festlegung des Abbildungs-/Beziehungstyps
- ▶ Festlegung der Kardinalitätsrestriktionen
- ▶ Festlegung der Relationship-Attribute mit Wertebereichen
- ▶ Evtl. Festlegung schwacher Entity-Typen

Abbildung 2.14 zeigt eine Übersicht der grafischen Elemente zur Darstellung von ER-Modellen.

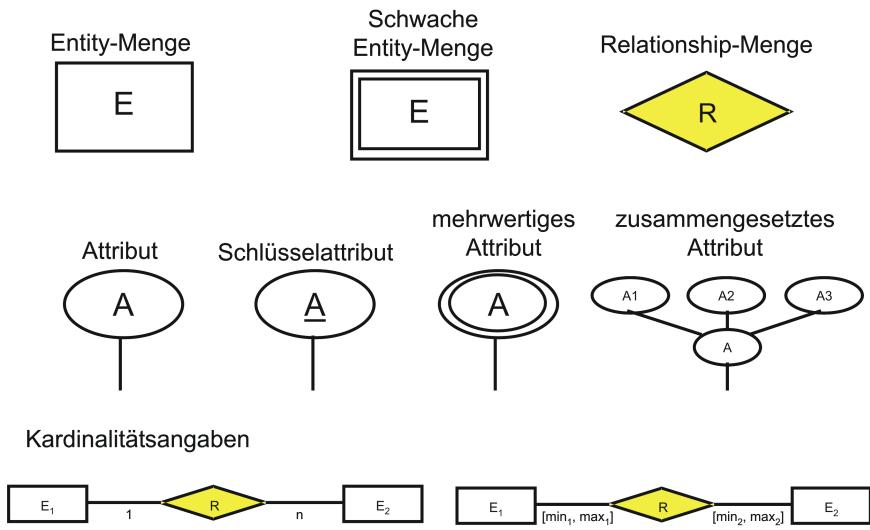


Abbildung 2.14: Übersicht der grafischen Symbole

2.3.3 Generalisierung

Die Generalisierung wird im konzeptionellen Entwurf eingesetzt, um eine natürlichere und übersichtlichere Strukturierung der Entity-Typen zu erzielen. Bei der Generalisierung werden die Eigenschaften ähnlicher Entity-Typen herausfaktorisiert und einem gemeinsamen Obertyp (Supertyp) zugeordnet. Die ähnlichen Entity-Typen heißen dann Untertypen (Subtypen) des generalisierten Obertyps.

Diejenigen Eigenschaften, die nicht faktorisierbar sind, da sie nicht in allen Untertypen vorkommen, verbleiben beim jeweiligen Untertyp. In dieser Hinsicht stellt der Untertyp eine Spezialisierung des Obertyps dar. Ein Schlüsselkonzept der Generalisierung ist die Vererbung: Ein Untertyp erbt sämtliche Eigenschaften des Obertyps. Dies bringt eine Reihe von Vorteilen:

- ▶ Keine Wiederholung von Beschreibungsinformationen
- ▶ Abgekürzte Beschreibung
- ▶ Fehlervermeidung

Die Entities eines Untertyps werden implizit auch als Entities des Obertyps betrachtet. Dies motiviert die Bezeichnung 'is-a' in der grafischen Darstellung von Generalisierungen. Da es sich hierbei um eine besondere Art der Beziehung handelt wird hierfür ein anderes grafisches Symbol, meist ein Dreieck, verwendet. Befindet sich die Obertyp auch in der Grafik oberhalb der Untertypen, kann auch auf ein Symbol verzichtet werden, man verbindet die Typen über eine Linie, die mit dem Beziehungsnamen 'is-a' gekennzeichnet ist (vgl. Abbildung 2.15).

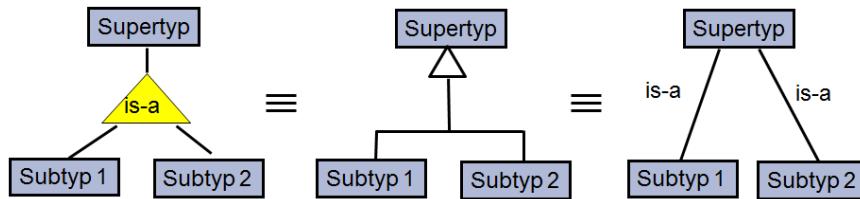


Abbildung 2.15: Darstellung von Generalisierungen

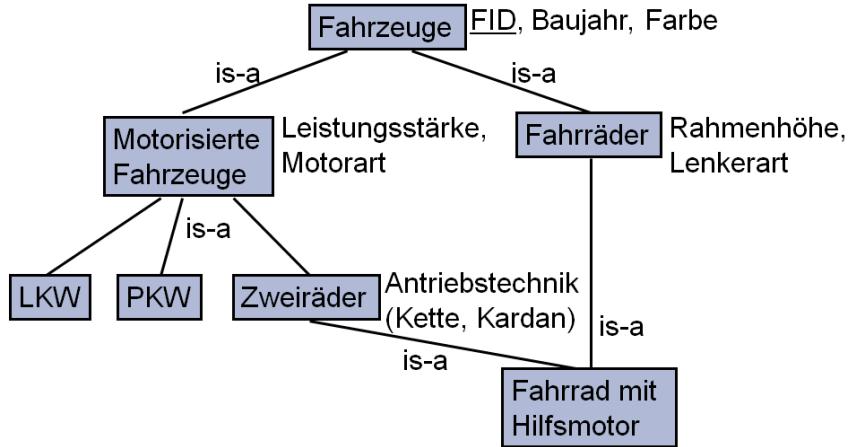


Abbildung 2.16: Beispiel einer Generalisierung

Jede Instanz des Untertyps ist immer auch Instanz des Obertyps, zwischen den Typen besteht eine implizite 1:1-Beziehung. Die Entitymenge eines Untertyps ist damit immer eine Teilmenge der Entitymenge des Obertyps. Hinsichtlich der Teilmengensicht sind bei der Generalisierung folgende Eigenschaften bedeutsam:

- ▶ Bei disjunkten Spezialisierungen sind die Entitymengen aller Untertypen eines Obertyps paarweise disjunkt (Kürzel D). Ansonsten ist die Spezialisierung überlappend (Ü).
- ▶ Eine Spezialisierung ist vollständig (V) (auch als total bezeichnet), wenn die Entitymenge des Obertyps keine direkten Elemente enthält, sich also nur aus der Vereinigung der Entitymengen der Untertypen ergibt. Ansonsten wird die Spezialisierung als partiell (P) bezeichnet.

Beispiele hierzu finden sich in Abbildung 2.18. Die Eigenschaften können auch direkt in das Diagramm über über die Anfangsbuchstaben der Merkmale integriert werden. Diese Form verwendet

man aber meist nur in Verbindung mit dem Dreieckssymbol zur Verdeutlichung der Vererbungsbeziehung. Für eine direkte Unterscheidung zwischen disjunkten und nicht disjunkten Vererbungen im Diagramm kann auch alternativ die in Abbildung 2.17 dargestellte Notation zu verwenden.

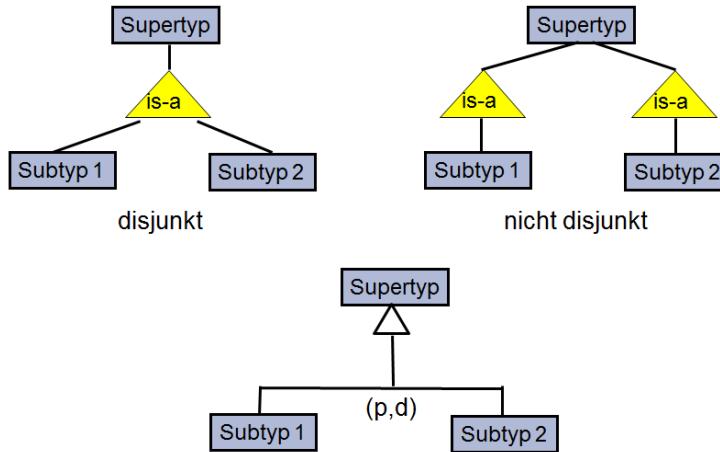


Abbildung 2.17: Darstellung disjunkter und nicht disjunkter Vererbungen

Beispiele:

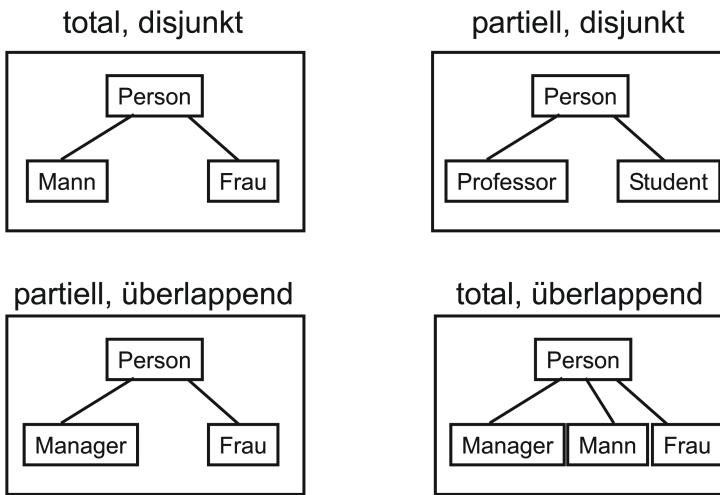


Abbildung 2.18: Beispiele zu Spezialisierungseigenschaften

2.3.4 Aggregation

Während man bei der Generalisierung gleichartige Entity-Typen strukturiert, werden bei der Aggregation unterschiedliche Entity-Typen, die in ihrer Gesamtheit einen strukturierten Objekttyp bilden, einander zugeordnet. In dieser Hinsicht kann man die Aggregation als einen besonderen Beziehungstyp deuten, der einem übergeordneten Entity-Typ mehrere untergeordnete Entity-Typen zuordnet. Diese Beziehung wird als Teil-von (part-of) bezeichnet, um zu betonen, dass die untergeordneten Entities Teile (also Komponenten) der übergeordneten, zusammengesetzten Entities sind. Finden sich die zusammengesetzten Entity-Typen in der Grafik oberhalb der Komponenten-Typen, kann auf die Raute als Beziehungstyp verzichtet werden (vgl. Abbildung 2.19). Ein Beispiel findet sich in Abbildung 2.20.

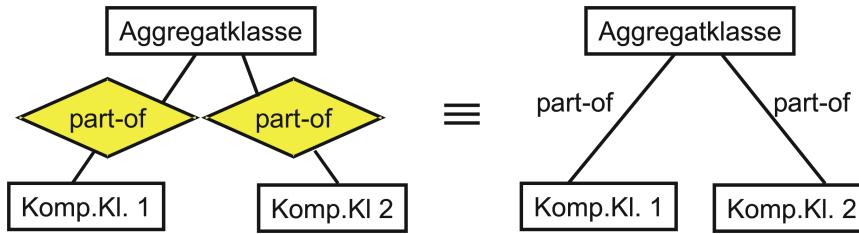


Abbildung 2.19: Darstellung von Aggregationen

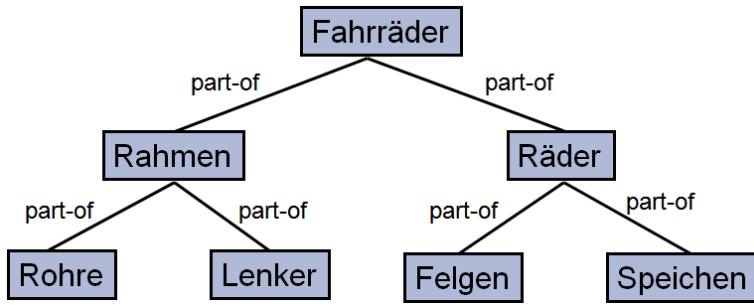


Abbildung 2.20: Beispiel einer Aggregation

Aggregationen haben folgende Eigenschaften:

- ▶ Unterstützung komplex-strukturierter Objekte.
- ▶ keine Vererbung.
- ▶ Subkomponentenelemente sind auch Elemente der Superkomponenten.
- ▶ Objekte können gleichzeitig Elemente verschiedener Komponenten bzw. Subkomponenten von mehreren Superkomponenten sein.

2.4 Konzeptioneller Entwurf mit dem ER-Modell

Wie geht man nun bei der Modellerstellung vor? Vorlage ist eine textuelle Beschreibung einer Aufgabe oder eines Vorgangs. Um aus ihr ein erstes Modell zu extrahieren, müssen zunächst Entitäten, Attribute und Beziehungen identifiziert werden. Dabei sind Substantive Kandidaten für Entitäten, Adjektive Kandidaten für Attribute und Verben Kandidaten für Beziehungen. Es ist jedoch Sorgfalt angebracht, denn auch Attribute können durch Substantive bezeichnet sein.

Entitätstypen, die aus einer solchen Betrachtung entstehen, müssen verifiziert werden. Besitzen sie keine Eigenschaften oder gehen sie keine Beziehung ein, sind sie vermutlich überflüssig. Umgekehrt ist es oft sinnvoll, eine Gruppe zusammengehöriger Attribute als eigenen Entitätstyp zu realisieren, wenn dieser für sich einen Sinn macht, Beziehungen eingehen kann und somit an mehreren Stellen benutzt werden kann. Auch hat sich folgende (sehr grobe) Vorgehensreihenfolge als empfehlenswert herausgestellt:

1. Entitäty-Typen bestimmen.
2. Relationships suchen.
3. Attribute suchen.

4. Kardinalität festlegen.

Im Laufe des Erstellungsprozesses ist das Modell regelmäßig auf Korrektheit und Redundanzfreiheit zu prüfen. Häufig passiert es, dass bei der Namensvergabe von Entity-Typen Singular und Plural gemischt wird. Auch die Verwendung von Synonymen ergibt häufig, dass zwei Entity-Typen für die gleichen Objekte erzeugt werden. Dies ist natürlich zu vermeiden. Umgekehrt muss bei Homonymen eine Trennung eines Typs in zwei Typen erfolgen.

Im Hinblick auf die in Abschnitt 2.1.2 genannten Qualitätsmerkmale verhält sich ein ER-Diagramm bzw. ein mit dem ER-Modell erstelltes konzeptionelles Schema wie folgt:

Vollständigkeit Dieses Merkmal ist offensichtlich schwierig zu überprüfen, denn es lässt sich nur durch einen möglichst genauen Vergleich mit der Anforderungsanalyse und dem erstellten ER-Diagramm auswerten. Dieser Abschlussvergleich sollte jedoch immer durchgeführt werden.

Korrektheit: Dieser Aspekt bedeutet in Bezug auf das ER-Modell, dass die Konstrukte dieses Modells richtig benutzt wurden. Es lassen sich dabei zwei Formen der Korrektheit unterscheiden:

1. Ein ER-Schema ist syntaktisch korrekt, wenn die im Schema enthaltenen Definitionen und Festlegungen das Modell nur in zulässiger Weise benutzen.
2. Ein ER-Schema ist semantisch korrekt, wenn die ER-Konzepte gemäß ihrer Definition angewendet wurden.

Häufige semantische Fehler sind eine Verwendung eines Attributs anstelle einer Entität, ein Vergessen einer Spezialisierung, ein Übersehen einer Vererbungseigenschaft, eine Verwendung einer Beziehung mit unzutreffender Anzahl beteiligter Entitäten, eine Nichtangabe eines Schlüssels.

Minimalität: Diese Eigenschaft lässt sich vergleichbar zur Vollständigkeit für ein ER-Diagramm nur auf informeller Ebene überprüfen.

Lesbarkeit: Diese Eigenschaft ist durch das ER-Modell mit seinen grafischen Symbolen zur Erstellung von Diagrammen in hohem Maße erfüllt. Allerdings kann man nur dann von wirklich guter Lesbarkeit sprechen, wenn ein ER-Diagramm auch ästhetischen Kriterien genügt.

1. Das Diagramm sollte auf eine Gitterstruktur gezeichnet werden.
2. Rechtecke und Rauten sollten gleich groß sein.
3. Obertypen in Hierarchien sollten auch oben stehen.
4. Das Diagramm sollte möglichst kreuzungsfrei sein.

Modifizierbarkeit: Ein Schema mit modularem Aufbau und guter Dokumentation ist im Allgemeinen gut modifizierbar und somit leicht an veränderte Gegebenheiten anpassbar. Für ein ER-Diagramm trifft dies speziell dann zu, wenn sich darin größere logische Einheiten leicht identifizieren lassen oder das Gesamtdiagramm sukzessive aus mehreren Teildiagrammen über wohldefinierte 'Schnittstellen' zusammengesetzt wird.

2.5 Ergänzungsübungen

Übung 2.2 Charakterisieren Sie 1:1, 1:n, n:1 und n:m-Beziehungen mittels der (min,max)-Notation.

Übung 2.3 Schwache Entity-Typen kann man immer auch als starke (normale) Entity-Typen modellieren. Was muss dabei beachtet werden?

Übung 2.4 Sie bekommen den Auftrag in einem Projektteam eine datenbankbasierte Anwendung für den Verkauf von Reisen zu entwickeln, die Informationen zu Reiseanbietern, Online-Shops, Reisebüros, Flughäfen und Hotels enthalten soll. Die Kernelemente der Datenbank sollen von Ihnen modelliert werden. Erstellen Sie hierzu aus folgenden Informationen ein ERM-Diagramm, das nur die Entity- und Relationship-Mengen sowie Beziehungstypen (1:1, 1:n, n:m) enthält. Attributen sind nicht im Diagramm darzustellen, diese sind textuell mit den Entity- und Relationship-Mengen aufzuführen. Die Angabe der Datentypen ist nicht erforderlich.

Reiseanbieter, die über einen eindeutigen Namen verfügen und an einem Ort ihren Hauptsitz haben, bieten jeweils Reisen zu verschiedenen Zielen und mit unterschiedlichen Reisedauern an, wobei hierfür jede Reise eine für den Anbieter eindeutige Kennung bekommt. Online-Reisevermittler, die einen für ein Land eindeutigen Namen haben, verkaufen auf ihren Web-Sites mit jeweils eigener URL diese Reisen an Kunden, die dabei den Starttermin sowie die Anzahl der Mitreisenden frei wählen können. An Kundeninformationen sind dabei der Name sowie seine Telefonnummer und Adresse sowie eine global eindeutige Kundenkennung abzuspeichern. Die Anbieter haben als Angestellte Reisebetreuer, die bei einzelnen Reisen mitreisen um diese vor Ort zu betreuen, sowie Verwaltungspersonal. Die Angestellten sollen in einer Vererbungshierarchie dargestellt werden, wobei jeder eine für den Anbieter eindeutige Personalnummer sowie einen Namen besitzt, Verwaltungsmitarbeiter jeweils einen Zuständigkeitsbereich haben und Reisebetreuer vielfältige Sprachkenntnisse besitzen können.

Kapitel 3

Das relationale Datenmodell

Eine der wesentlichen Grundlagen zur Verwaltung von Daten wurden mit der Entwicklung des relationalen Datenmodells durch Edgar F. 'Ted' Codd und eine prototypische Implementierung des Modells durch eine IBM-Gruppe (System-R) gelegt. Der britische Mathematiker Edgar F. Codd (1923-2003), von 1949-1979 IBM-Mitarbeiter und IBM-Fellow und 1981 für seine wegweisenden Arbeiten mit dem ACM-Turing-Award geehrt, konnte mit seiner Arbeit 'A relational model of data for large shared data banks' nur geringe Aufmerksamkeit erzielen.

Sein Modell fand jedoch Unterstützung durch eine kleine, aber einflussreiche Gruppe an exzellenten Technologen innerhalb der IBM, die die Coddsche Vision verstanden. Durch eine Implementierung des relationalen Modells in dem prototypischen relationalen Datenbanksystem System-R zeigten sie, dass dies auch effizient mit den geforderten Eigenschaften realisiert werden konnte. Gleichzeitig wurde das Relationenmodell von vielen, mehr theoretisch orientierten Forschern als geeignete Grundlage wahr genommen, das Modell weiter auf Eigenschaften zu prüfen, neue Konzepte hinzuzufügen und damit das Verständnis für das relationale Modell zu vertiefen.

Sein Ansatz für ein Datenmodell war einfach und mathematisch fundiert. Die Besonderheit dieses Datenmodells besteht in der mengenorientierten Verarbeitung der Daten im Gegensatz zu den bis dahin vorherrschenden satzorientierten Datenmodellen wie das Netzwerkmodell und das hierarchische Datenmodell. Das relationale Datenmodell ist im Vergleich zu den satzorientierten Modellen sehr einfach strukturiert. Diese Einfachheit hat wesentlich zu dem Erfolg des Modells beigetragen. Eine erste prototypische Umsetzung schuf Mitte der 70er Jahre des 20. Jahrhunderts IBM mit dem System R, Oracle war der erste Anbieter einer kommerziellen Implementierung. Seit Mitte der 80er Jahre ist das relationale Datenmodell das Standardmodell kommerzieller Datenbanksysteme.

3.1 Relationen und Relationenschemata

Im relationalen Datenmodell werden Objekttypen der Anwendung durch Relationenschemata beschrieben. Diese bestehen aus einer Menge von Attributen, die die gemeinsamen Eigenschaften der Objekte repräsentieren, die zu einem darstellbaren Objekttyp gehören. Attributen werden Wertebereiche (domains) zugeordnet, die in der Praxis meist einem Standardtyp wie Integer, String, Real oder Double entsprechen. Analog zum Entity-Relationship-Modell werden auch hier die Wertebereiche um den Wert NULL erweitert, der für 'unbekannt' oder 'nicht existent' stehen kann.

Definition 3.1 Gegeben sei eine Menge von Wertebereichen primitiver Datentypen $D = \{D_1, \dots, D_n\}$, die als Domains bezeichnet werden. Eine Relationenschema $S = \{A_1, \dots, A_m\}$, besteht aus einer Menge von Attributnamen, wobei für jedes A_i ein Domain aus D festzulegen ist.

Definition 3.2 Ein Datenbankschema besteht aus einer Menge von Relationenschemata.

Eine Relation ist nun in ihrer einfachsten Beschreibung eine Teilmenge des kartesischen Produkts über den Wertebereichen der Attribute des Relationenschemas. Sie stellt die zu einem Relationenschema passenden und aktuell vorhandenen Daten dar – die Ausprägung zu diesem Schema. Alternativ kann man Relationen über die Elemente definieren:

Definition 3.3 Sei $S = \{A_1, \dots, A_m\}$ ein Relationenschema. Eine Relation R über S ist eine endliche Menge von Tupeln, wobei jedes Tupel aus Werten $\{w_1, \dots, w_m\}$ besteht, die jeweils im Wertebereich des zugehörigen Attributs liegen.

Für den Wert eines Tupels t bezüglich eines Attributs A_i schreibt man $t.A_i$ oder $t[A_i]$.

Eine Relation kann anschaulich als Tabelle verstanden werden. Die Attribute des Relationenschemas bilden die Spaltenüberschriften der Tabelle, die Tupel sind die Zeilen. Alle Tabelleneinträge in den Zeilen passen dabei immer zu den definierten Wertebereichen der Attribute. Eine Veranschaulichung dieser Begriffe wird in Abbildung 3.1 gegeben.

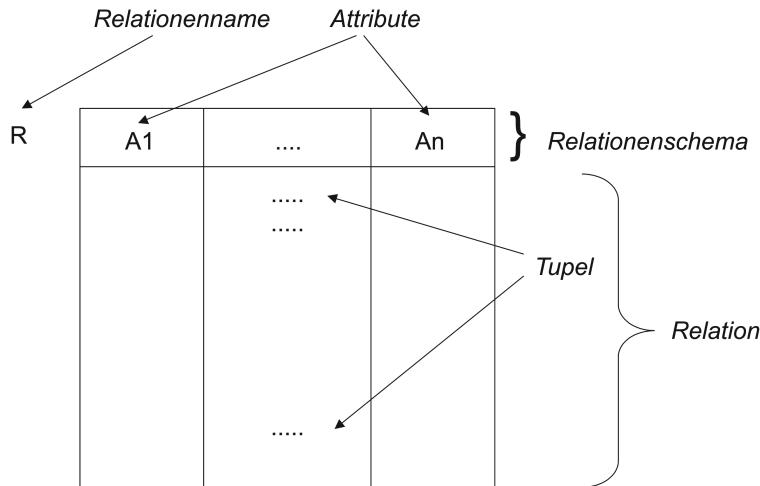


Abbildung 3.1: Veranschaulichung der Begriffe

An den verschiedenen Tabellenpositionen können somit nur 'atomare' Attributwerte stehen und nicht wiederum Relationen oder andere strukturierte Werte. Diese Einschränkung nennt man auch die erste Normalform (1NF) für Relationen. 1NF-Relationen werden oft auch als flache Relationen bezeichnet.

Auch wenn der Begriff Tabelle für eine Relation in der Praxis sehr häufig verwendet wird, ist dies formal betrachtet nicht korrekt. Der mathematische Begriff Relation steht für eine Menge. Mengen haben nun zwei zentrale Eigenschaften, die auch für das relationale Datenbankmodell gelten. So gibt es zum einen innerhalb einer Menge keine doppelten Elemente. In einer Tabelle kann man auch Zeilen mehrfach aufnehmen. Zum anderen gibt es keine Reihenfolge der Elemente einer Relation. Dies ist beim Umgang mit relationalen Datenbanksystemen zu beachten, die Reihenfolge der Datensätze, die ein relationales Datenbanksystem bei einer Anfrage zurückgibt, ist frei und kann vom Datenbanksystem bestimmt werden. Insbesondere kann sich die Reihenfolge bei der nächsten Anfrage ändern. Dies ist zulässig, da die Reihenfolge aufgrund der Mengeneigenschaft ohne Bedeutung ist. Im relationalen Datenmodell ist, wenn auch im Widerspruch zu dem Begriff Relation, die Reihenfolge der Spalten ohne Bedeutung. Zusammenfassend haben Relationen in Tabellendarstellung damit folgende Eigenschaften (vg. hierzu auch Abbildung 3.2):

- Jede Zeile ist eindeutig und beschreibt ein Objekt der Miniwelt.

Studiengänge		Studenten			
SG	Name	MNr	Name	SB	SG
TIT	Informationstechnik	1532	Geisler	2008	WWI
WWI	Wirtschaftsinformatik	4577	Abel	2008	WWI

Abbildung 3.2: Beispiel einer Tabellendarstellung von Relationen

- ▶ Die Ordnung der Zeilen ist ohne Bedeutung.
- ▶ Die Ordnung der Spalten ist ohne Bedeutung, der Zugriff erfolgt nur über den Attributnamen.
- ▶ Jeder Datenwert ist atomar.
- ▶ Relationen sind die einzige Datenstruktur, alle für den Benutzer bedeutungsvollen Informationen sind ausschließlich durch Datenwerte in Relationen ausgedrückt.

Definition 3.4 Sei $R = \{R_1, \dots, R_n\}$ eine endliche Menge von Relationenschemata in 1NF. Eine relationale Datenbank d über R ist eine Menge von Relationen zu den Relationenschemata.

3.2 Modellinhärente Integritätsbedingungen

Da eine Relation eine Menge ist, können keine zwei Tupel mit identischen Werten für alle Attribute eines Relationenschemas in dieser Relation existieren. Sonst ist aber bisher alles erlaubt.

3.2.1 Primärschlüssel

Genau wie im Entity-Relationship-Modell müssen identifizierende Attributmengen für Relationenschemata angegeben werden, um eine gewisse Konsistenz in den zugehörigen Datensätzen sicherzustellen. Die entsprechenden Attributwerte identifizieren dann jedes Tupel aus einer Relation eindeutig. Sind die Attributmengen bzgl. der Teilmengenbeziehung \subseteq minimal gewählt, so bezeichnet man diese als Schlüsselkandidat für das Relationenschema. Ein Schlüsselkandidat muss speziell ausgezeichnet sein, dies ist dann der Primärschlüssel. Da der Primärschlüssel zur Identifikation eines Datensatzes dient, darf dieser nie den Wert NULL annehmen. Diese Aussagen führen zu folgender Definition:

Definition 3.5 Eine identifizierende Attributmenge (Schlüsselkandidat) für eine Relation R zu einem Relationenschema

$S = \{A_1, \dots, A_n\}$ ist eine Menge $K = \{K_1, \dots, K_m\} \subseteq S$, so dass gilt:

- ▶ $\forall t \in R, \forall A \in K$ gilt: $t[A] \neq \text{NULL}$
- ▶ $\forall t_1, t_2 \in R$ mit $t_1 \neq t_2$ gilt: $\exists A \in K$ mit $t_1[A] \neq t_2[A]$
- ▶ K ist eine bezüglich \subseteq minimale, identifizierende Attributmenge.

Primärattribut nennt man jedes Attribut eines Schlüsselkandidaten. Ein Primärschlüssel ist ein ausgezeichneter Schlüsselkandidat.

Primärschlüsselattribute werden durch unterstreichen der Attribute gekennzeichnet:

Beispiele:

- ⇒ Studenten(MatNr, Name, Studienbeginn, SG)
- ⇒ Studiengänge(Kürzel, Name)
- ⇒ Lieferung(Lieferant, Lieferdatum, Teil, Anzahl)

3.2.2 Fremdschlüssel

Betrachtet man die Tabellen in Abbildung 3.2, so fällt auf, dass die Werte der Spalte SG in der Studentenrelation zu Werten des Primärschlüssels der Studiengangsrelation passen. Über die Gleichheit der Werte wird hierbei ausgedrückt, dass der Student Abel den Studiengang Wirtschaftsinformatik studiert. Diese Zuordnung kann jedoch nur durchgeführt werden, wenn der Wert der SG-Spalte eines Studenten auch in der Studiengangsrelation vorhanden ist. Dies kann im relationalen Datenbankmodell über Fremdschlüssel gewährleistet werden.

Definition 3.6 Eine Attributliste X in einem Relationenschema R_1 ist ein Fremdschlüssel, wenn in einem Relationenschema R_2 eine kompatible Attributliste Y Primärschlüssel (oder Schlüsselkandidat) ist und die Attributwerte zu X im Falle $\neq \text{NULL}$ auch in den entsprechenden Spalten der Relation zu R_2 vorkommen.

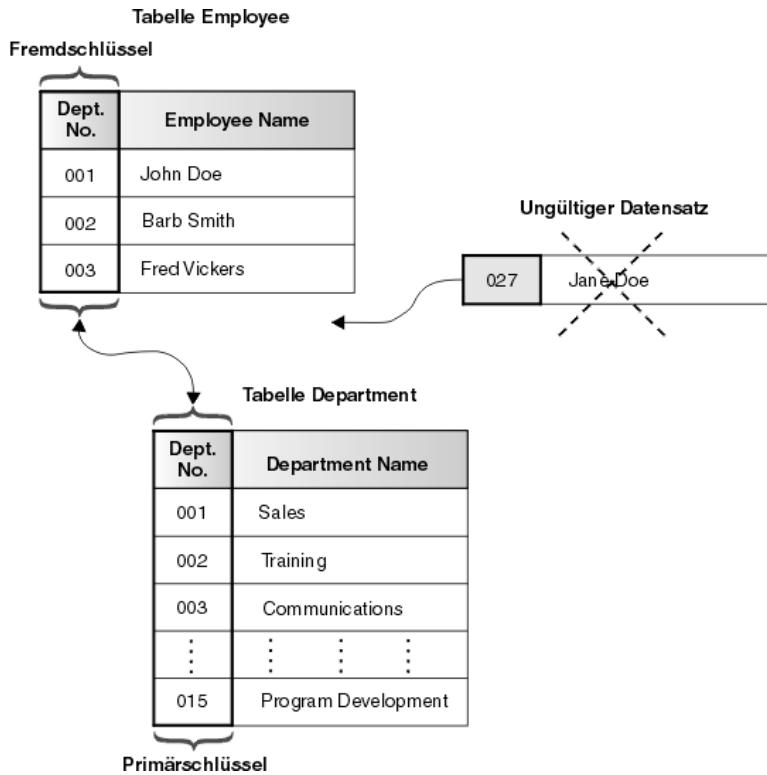


Abbildung 3.3: Beispiel einer Fremdschlüsselprüfung

Ein Fremdschlüssel ist dabei zusammengesetzt, wenn der dazugehörige Primärschlüssel zusammengesetzt ist. Fremdschlüssel und zugehöriger Primärschlüssel (Schlüsselkandidat) sind auf dem gleichen Wertebereich definiert und damit vergleichbar. Sie gestatten die Verknüpfung von Relationen mit Hilfe von Relationenoperationen. Fremdschlüssel können NULL-Werte aufweisen.

Eine Relation kann mehrere Fremdschlüsselelemente besitzen, die auf gleiche oder verschiedene Relationen referenzieren. Auch Selbstreferenzierungen und Zyklen sind möglich. Beispiele hierfür werden im nächsten Abschnitt vorgestellt. Möchte man eine Fremdschlüsselbeziehung zwischen zwei Relationschemata darstellen, so kann man die Referenzierung über einen Pfeil von der Fremdschlüsselrelation zu der referenzierten Relation darstellen (vgl. Abbildung 3.4) oder durch Angabe der Referenzierung im Schema verdeutlichen:

Beispiel: Studenten (MNR, Name, SB, SG REFERENCES Studiengänge)



Abbildung 3.4: Pfeildarstellung einer Fremdschlüsselbeziehung

Ein relationales Datenbanksystem gewährleistet immer, dass eine gegebene Fremdschlüsselbeziehung erfüllt wird. Hierzu muss das Datenbanksystem bei Änderungen an Fremdschlüsselattributen oder referenzierten Primärschlüsselattributen den Datenbestand kontrollieren. Betrachtet man das Studentenbeispiel aus Abbildung 3.2, so kann man drei prinzipielle Änderungssituationen erkennen:

Einfügen eines Studiengangs oder Löschen eines Studenten

Hier kann die referentielle Integrität nie verletzt werden, bzgl. der Fremdschlüsselbeziehung sind diese Vorgänge immer möglich, wobei natürlich andere Integritätsbedingungen auch erfüllt sein müssen.

Einfügen oder Studiengangsänderung eines Studenten

Diese Operationen sind nur möglich, wenn der neue Fremdschlüsselwert auch in der referenzierten Relation existiert, ansonsten ist der Vorgang vom Datenbanksystem abzulehnen und mit einer entsprechenden Meldung zu quittieren.

Löschen oder Primärschlüsseländerung eines Studiengangs

Prinzipiell muss nach der Operation der Datenbankzustand konsistent sein, die referentielle Integrität muss bewahrt werden. Sofern kein Student den betroffenen Studiengang referenziert, ist dies immer gegeben. Existiert jedoch ein referenzierender Student, kann die Integrität auf verschiedene Arten gewährleistet werden. Im SQL92-Standard wird hierzu die Möglichkeit gegeben, Lösch- und Änderungsregeln bei der Fremdschlüsselangabe festzulegen, die das Verhalten steuern. Folgende Alternativen werden dabei angeboten, die alle die referentielle Integrität gewährleisten.

NOACTION: Existiert ein referenzierender Datensatz, wird die Operation mit einer Fehlermeldung abgebrochen, der alte konsistente Zustand bleibt erhalten. Dies ist die Defaulteinstellung.

CASCADE: Änderungen am referenzierten Wert werden auf die referenzierenden Werte übertragen. Das Löschen des referenzierten Satzes hat das Löschen der referenzierenden zur Folge.

SET NULL: Änderungen des referenzierten Werts bewirken einen Eintrag von NULL bei den Fremdschlüsselattributen der referenzierenden Sätze.

SET DEFAULT: Änderungen des referenzierten Werts bewirken einen Eintrag der Defaultwerte bei den Fremdschlüsselattributen der referenzierenden Sätze.

Der Einsatz dieser Lösch- und Änderungsregeln ist jedoch sorgfältig zu planen. Zum einen muss die verwendete Regel auch für die Problemstellung sinnvoll sein, zum anderen muss auf Abhängigkeiten zu anderen Regeln geachtet werden. Man betrachte hierzu das Beispiel in Abbildung 3.5, wobei NA für NOACTION und C für CASCADE steht.

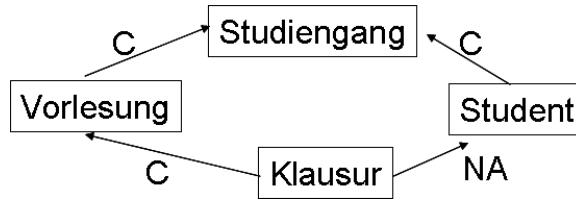


Abbildung 3.5: Falscher Einsatz von Löschregeln

Wird hier beispielsweise der Studiengang Tourismus gelöscht, so werden durch die linken Regeln die Vorlesungen und Prüfungen mitgelöscht. Danach können auch die Studenten gelöscht werden, da diese keine Prüfungen aufgrund der vorherigen Löschung derselben besitzen. Dadurch wird alles, was mit Tourismus zu tun hat, aus der Datenbank entfernt. Prüft hingegen das Datenbanksystem zuerst die rechte Löschregel, so wird versucht die Studenten zu löschen. Dies scheitert, da Prüfungen vorhanden sind, der gesamte Löschvorgang wird abgebrochen, alles vom Tourismus bleibt erhalten. Welche Variante nun tatsächlich ausgeführt wird, ist nicht definiert. Deshalb ist eine solche Definition beim Design zu vermeiden:

Definition 3.7 R_n ist löschabhängig von R_1 gdw. ein DELETE auf R_1 den Inhalt von R_n beeinflusst.

Wenn R_n löschabhängig von R_1 über zwei oder mehr referentielle Pfade ist, dann muss jeder Fremdschlüssel in R_n , der zu diesen Pfaden gehört, dieselbe Löschregel besitzen.

Abschließend gibt Tabelle 3.1 eine Übersicht über die in diesem Abschnitt definierten Begriffe.

Begriff	Informelle Bedeutung
Attribut	Spalte einer Tabelle
Wertebereich	mögliche Werte eines Attributs
Attributwert	Element eines Wertebereichs
Relationenschema	Menge von Attributen
Relation	Menge von Zeilen einer Tabelle
Tupel	Zeile einer Tabelle
Datenbankschema	Menge von Relationenschemata
Schlüsselkandidat	minimale identifizierende Attributmenge
Primärschlüssel	ausgezeichneter Schlüsselkandidat
Fremdschlüssel	referenzierende Attributmenge

Tabelle 3.1: Begriffsübersicht

3.3 Die zwölf RDBMS-Regeln

Im Oktober 1985 veröffentlichte Codd zwölf Regeln, nach denen ein System nur dann als vollständig relational angesehen werden kann, wenn es alle in den Regeln formulierten Anforderungen erfüllt. Eine fundamentale Grundregel formuliert er dabei als Regel 0:

Jedes System, das als relationales Datenbankmanagementsystem bezeichnet wird, muss in der Lage sein, die gesamte Datenbank mit seinen relationalen Fähigkeiten, wie im relationalen Modell spezifiziert, zu verwalten.

Die einzelnen Regeln sind:

1. Alle Informationen in der Datenbank werden in Relationen gespeichert.
2. Der Zugriff auf die Daten wird durch Angabe des Tabellennamens und des Spaltennamens möglich.
3. Es werden NULL-Werte unterstützt.
4. Alle Angaben über die Datenbank selbst (Metadaten) werden ebenfalls in Tabellen gespeichert.
5. Es gibt eine einheitliche Datenbanksprache.
6. Daten sollen durch eine View (Sicht auf eine Datenteilmenge) änderbar sein.
7. Man soll mit einzelnen Datensätzen als auch mit Mengen von Datensätzen arbeiten können (mengenorientierte Verarbeitung).
8. Daten und Anwendungsprogramme sind unabhängig von Änderungen der physikalischen Datenstruktur (physikalische Datenunabhängigkeit).
9. Es soll möglich sein, die logische Struktur zu ändern, ohne eine Anwendung ändern zu müssen (logische Datenunabhängigkeit).
10. Integritätsprüfung der Daten erfolgt durch das DBMS und nicht durch die Anwendungsprogramme.
11. Der Anwender muss keine Kenntnis darüber haben, wo und wie seine Daten gespeichert sind.
12. Es darf nicht möglich sein, alle bisher genannten Regeln zu unterwandern.

Neben diesen Regeln von Codd haben auch eine Reihe anderer Autoren vergleichbare Regeln und Kriterienkataloge aufgestellt.

3.4 Überführung von ER-Diagrammen in das Relationenmodell

Das Entity-Relationship Modell besitzt zwei grundlegende Strukturierungskonzepte:

- ▶ Entity-Typen und
- ▶ Beziehungstypen.

Dem steht im relationalen Modell nur ein einziges Strukturierungskonzept – die Relation – gegenüber. Also werden sowohl Entity-Typen als auch Beziehungstypen jeweils auf Relationen abgebildet. Dabei sind folgende Anforderungen einzuhalten:

- ▶ Informationserhaltung
- ▶ Minimierung der Redundanz
- ▶ Minimierung des Verknüpfungsaufwands
- ▶ Natürlichkeit der Abbildung
- ▶ Keine Vermischung von Objekten
- ▶ Verständlichkeit

3.4.1 Relationale Darstellung von Entity-Typen

Die relationale Darstellung von Entity-Typen kann für Entity-Typen ohne mehrwertige und zusammengesetzte Attribute direkt angegeben werden. Für jeden Entity-Typ der Form $E = (X)$ wird ein eigenes Relationenschema eingeführt, wobei der Name und der Primärschlüssel aus der Entity-Deklaration übernommen wird.

Da das flache Relationenmodell weder mehrwertige noch zusammengesetzte Attribute kennt, müssen diese auf einwertige Attribute zurückgeführt werden. Es sei bemerkt, dass man dadurch keine Ausdruckskraft verliert, nur die Angemessenheit der Darstellung ist zu klären.

Bei zusammengesetzten Attributen wird dieses durch seine Komponenten, jetzt aufgefasst als Einzelattribute, ersetzt. Im ER-Diagramm aus Abbildung 2.3 ist beispielsweise das Attribut Verlag des Entity-Typs Buch aus Name und Ort zusammengesetzt. Diese Zusammensetzung wird in der erzeugten Relation durch Name und Ort ersetzt.

Mehrwertige Attribute erfordern demgegenüber die Einführung neuer Relationenschemata. Jedes einzelne mehrwertige Attribut ist durch ein neues Schema darzustellen, welches als Attribute das mehrwertige sowie die Schlüsselattribute des Ausgangs-Entity-Typs enthält. Der logische Zusammenhang zwischen diesen Relationenschemata wird durch den Schlüssel des Ausgangs-Entity-Typs hergestellt, über den ein Wert des mehrwertigen Attributs einem Datensatz der Bezugsrelation zugeordnet werden kann. Die Schlüsselattribute des Ausgangs-Entity-Typs sind damit im Relationenschema des mehrwertigen Attributs Fremdschlüssel. Der Primärschlüssel ergibt sich aus dem Fremdschlüssel und dem zuvor mehrwertigen Attribut.

Das Buch-Beispiel aus Abbildung 2.3 lässt sich damit wie folgt als relationales Schema darstellen:

Buch (InvNr, Titel, VerlagName, VerlagOrt, E-Jahr)

BuchAutoren (InvNr REFERENCES Buch, Autor)

Beim Herunterkopieren der Attribute des zusammengesetzten Attributs Verlag wurde hier der Hauptattributname *Verlag* vor die Unterattribute *Name* und *Ort* gestellt. Diese Vorgehensweise ist jedoch nicht verpflichtend, Attributnamen können auch direkt übernommen werden, sofern das Verständnis der resultierenden Relation gegeben ist.

3.4.2 Relationale Darstellung von Beziehungen

Auch Relationship-Typen müssen mit Hilfe von Relationen dargestellt werden. Dabei kann man die Beziehungsinformation bei den Relationenschemata der beteiligten Entities darstellen oder die Beziehung durch ein eigenes Relationenschema repräsentieren. Welche Form möglich und sinnvoll ist, hängt primär vom Beziehungstyp ab.

Die Darstellung einer Beziehung als eigenes Relationenschema ist immer möglich (aber nicht immer sinnvoll). Dabei umfasst das erzeugte Relationenschema alle Primärschlüsselattribute der beteiligten Entity-Typen. Diese sind jeweils Fremdschlüssel auf die Relationenschemata der dazugehörigen Entity-Typen. Zusätzlich werden die Attribute der Beziehung in das Schema aufgenommen. Der Primärschlüssel ist vom Beziehungstyp abhängig. Im folgenden werden die verschiedenen Beziehungstypen und die Umsetzungsmöglichkeiten betrachtet.

3.4.2.1 1:n - Beziehungen

Betrachtet man das Beispiel in Abbildung 3.6, so sind die beiden Entity-Typen in jeweils eine eigene Relation zu überführen. Wie zuvor beschrieben, kann die Beziehung 'Abteilungszugehörigkeit' in eine eigene Relation überführt werden.



Abbildung 3.6: Umsetzung einer 1:n-Beziehung

- ▶ ABT (ANR, ANAME, ...)
- ▶ PERS (PNR, PNAME, ...)
- ▶ ABT-ZUGEH (PNR REFERENCES PERS, ANR REFERENCES ABT)

Bei einer 1:n-Beziehung reicht es aus, dass in der Relation für die Beziehung der n-Teil Primärschlüssel wird, da dieser immer nur einmal in der Relation auftaucht. In dem Beispiel kann ein Mitarbeiter nur einmal in der Relation 'ABT-ZUGEH' vorkommen, da er nur zu maximal einer Abteilung gehört. Gibt es keine Abteilungszugehörigkeit eines Mitarbeiters (minimale Kardinalität ist 0), gibt es keinen Eintrag in der Relation.

1:n-Beziehungen können jedoch kompakter überführt werden, indem in der Relation, die maximal einen Beziehungspartner hat (hier PERS), auf die andere Relation verwiesen wird.

- ▶ ABT (ANR, ANAME, ...)
- ▶ PERS (PNR, PNAME, ..., ANR REFERENCES ABT)

Gibt es keine Abteilungszugehörigkeit eines Mitarbeiters, wird bei ANR NULL eingetragen. NULL-Werte bei Fremdschlüsseln sind zulässig. Diese Überführung wird meist aufgrund der Einfachheit der Darstellung bevorzugt. Hat jedoch die Beziehung eigene Attribute, kann die Darstellung in einer eigenen Relation für die Beziehung anschaulicher sein.

3.4.2.2 1:1 - Beziehungen

Eine 1:1-Beziehung kann analog umgesetzt werden. Für das Beispiel in Abbildung 3.7 sieht die Umsetzung der Relation in eine eigene Relation wie folgt aus:



Abbildung 3.7: Umsetzung einer 1:1-Beziehung

- ▶ PERS (PNR, PNAME, ...)
- ▶ EHE (PNR1 REFERENCES PERS,
PNR2 REFERENCES PERS)

In der Beziehungsrelation können beide beteiligten Personen nur maximal einmal auftauchen. Deshalb sind beide Fremdschlüssel (die übernommenen Primärschlüssel der beteiligten Entity-Typen) mögliche Schlüsselkandidaten, von denen einer als Primärschlüssel herausgesucht werden kann. Aber auch hier gibt es eine kompaktere Darstellung, da aufgrund der maximalen Anzahl von

eins an Beziehungspartnern in der Relation der Entity-Typen direkt auf diesen verwiesen werden kann. Dies gilt für beide Entity-Mengen. Da in dem Beispiel eine rekursive Beziehung vorliegt, muss nur diese eine Relation die Beziehung aufnehmen. Auch hier kann der Fremdschlüssel NULL sein.

- ▶ PERS (PNR, PNAME, ..., PNR-GATTE REFERENCES PERS)

3.4.2.3 n:m - Beziehungen

Da bei n:m-Beziehungen Entities jeweils mit mehreren Partnern in Beziehung stehen können, ist ein direkter Verweis auf die Partner in den Relationen zu den Entities nicht möglich. Deshalb muss bei einer n:m-Beziehung diese in einer eigenen Relation dargestellt werden.¹ Für das Beispiel in Abbildung 3.7 sieht die Umsetzung der Relationship in eine eigene Relation wie folgt aus:

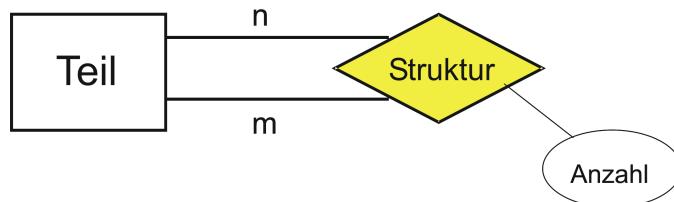


Abbildung 3.8: Umsetzung einer n:m-Beziehung

- ▶ TEIL (TNR, BEZEICHNUNG, MATERIAL, BESTAND)
- ▶ STRUKTUR (ONR REFERENCES Teil, UNR REFERENCES Teil, Anzahl)

Da in der Beziehungstabelle die referenzierten Teile mehrfach vorkommen können, ist der Primärschlüssel zusammengesetzt aus den Primärschlüsseln beider referenzierten Relationen. Abbildung 3.9 zeige die Relationen mit Inhalt für eine Beispielsituation.

3.4.2.4 Beziehungen mit Grad größer zwei

Stellt eine Relationship eine Beziehung zwischen mehr als zwei Entities her, wird dies wie eine n:m-Beziehung behandelt. Aus der Beziehung wird eine eigene Relation erzeugt, die die Primärschlüssel der referenzierten Entity-Menge enthält, wobei diese gemeinsam den Primärschlüssel bilden. Dies sieht für das Beispiel in Abbildung 3.10 wie folgt aus:

- ▶ LIEFERANT (LNR, LNAME, LORT, ...)
- ▶ PROJEKT (PRONR, PRONAME, PROORT, ...)
- ▶ TEIL (TNR, TBEZ, GEWICHT, ...)
- ▶ LIEFERUNG (LNR REFERENCES LIEFERANT, PRONR REFERENCES PROJEKT, TNR REFERENCES TEIL, ANZAHL, DATUM)

¹Die Beziehung wird dadurch aufgespalten in eine 1:n-Beziehung zur Beziehungsrelation und von dort mit einer n:1-Beziehung zur anderen Entity-Relation.

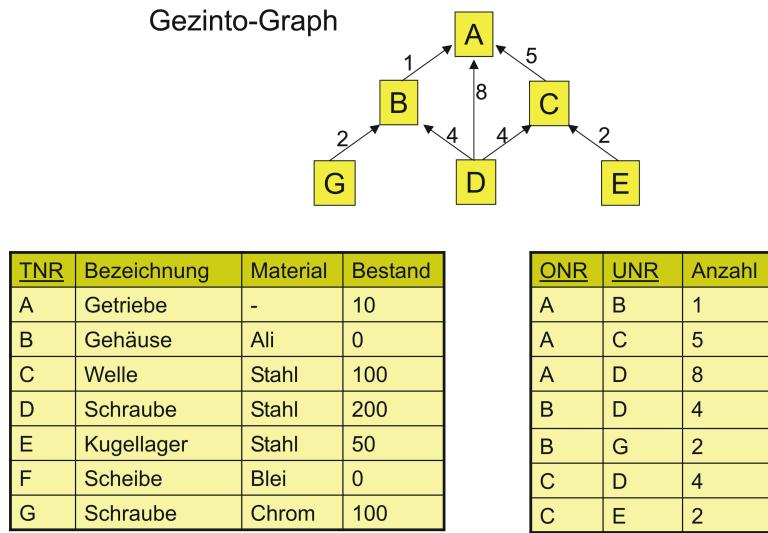


Abbildung 3.9: Beispiel einer n:m-Beziehungsumsetzung

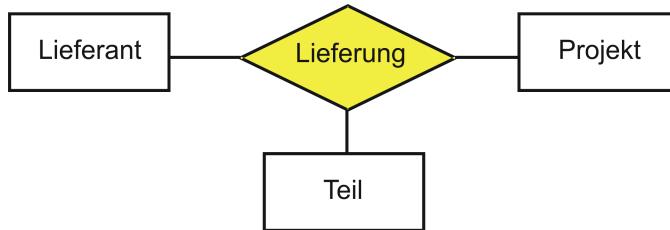


Abbildung 3.10: Beziehungen mit Kardinalität größer zwei

3.4.2.5 Abbildung schwacher Entity-Typen

Schwache Entity-Typen sind ein Spezialfall der 1:n-Beziehung. Auch hier kann der n-Teil (das abhängige Entity) um den Primärschlüssel des Entity-Typs, von dem die Abhängigkeit besteht, erweitert werden, wodurch die Beziehung hergestellt wird. Dieser wird hier jedoch in den Primärschlüssel aufgenommen, da die Tupel erst über den übergeordneten Typ eindeutig identifiziert werden können. Für das Beispiel in Abbildung 3.11 sieht dies Umsetzung wie folgt aus:

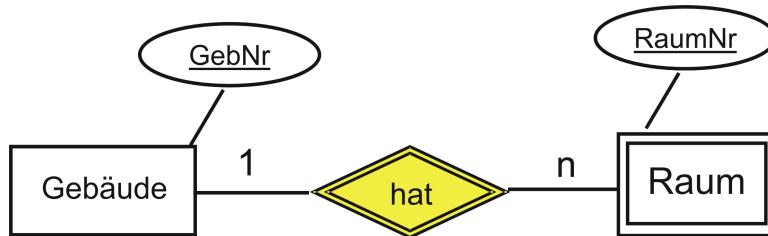


Abbildung 3.11: Beispiel schwacher Entity-Typen

- ▶ Gebäude (GebNr, ...)
- ▶ Raum (RaumNr, GebNr REFERENCES Gebäude, ...)

3.4.2.6 Zusammenfassung der Umsetzungsregeln

Zusammenfassend ergeben sich folgende Umsetzungsregeln:

- ▶ Aus Entity-Typen wird immer eine eigene Relation.
- ▶ Mehrwertige Attribute werden durch eine eigene Relation dargestellt.
- ▶ Zusammengesetzte Attribute werden runtergebrochen.
- ▶ 1:n- und 1:1-Beziehungen lassen sich ohne eigene Relation darstellen. Hierzu wird in der Relation, der maximal ein Tupel der anderen zugeordnet ist, der Primärschlüssel der referenzierten Relation als Fremdschlüssel verwendet.
- ▶ Ein n:m-Relationship-Typ muss durch eine eigene Relation dargestellt werden. Die Primärschlüssele der zugehörigen Entity-Typen treten als Fremdschlüssele auf, die gemeinsam den Primärschlüssel der Beziehungsrelation bilden.
- ▶ Bei schwachen Entity-Mengen muss der Primärschlüssel erweitert werden.

Auch Aggregationsbeziehungen können anhand dieser Umsetzungsregeln behandelt werden.

3.4.3 Abbildung von Generalisierung im Relationenmodell

Das Relationenmodell sieht keine Unterstützung der Vererbung vor. Deshalb muss die Umsetzung simuliert werden, wobei dies nur eingeschränkt möglich ist. Im folgenden werden vier Möglichkeiten der Umsetzung dargestellt, die alle ihre Vor- und Nachteile haben. Diese Alternativen werden anhand des in Abbildung 3.12 dargestellten Beispiels beschrieben.

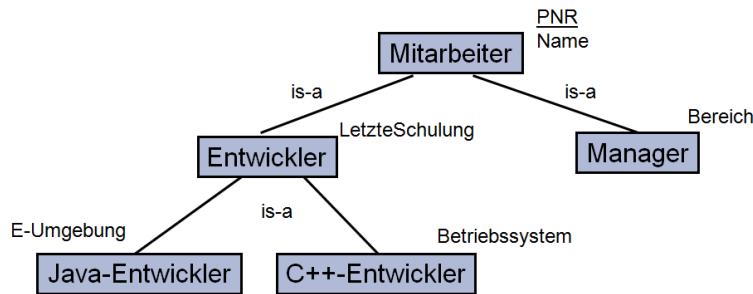


Abbildung 3.12: Beispiel einer Generalisierung

3.4.3.1 Hausklassenmodell – Horizontale Abbildung

Beim Hausklassenmodell wird die Vererbung per Hand durchgeführt. Für jeden Entity-Typ wird eine Relation erzeugt, die alle Attribute einschließlich der ererbten besitzt.

- ▶ Mitarbeiter (PNR, Name)
- ▶ Manager (PNR, Name, Bereich)
- ▶ Entwickler (PNR, Name, LetzteSchulung)
- ▶ Java-Entwickler (PNR, Name, LetzteSchulung, EU)

- ▶ C++-Entwickler (PNR, Name, LetzteSchulung, BS)

Jede Instanz ist genau einmal und vollständig in ihrer Hausklasse gespeichert. Dadurch kann bei totalen Ableitung die Oberklasse auch komplett entfallen, hierfür gibt es keine abzulegenden Tupel.

Der Vorteil dieser Lösung ist eine Speicherplatz sparende Umsetzung, da Redundanzen vermieden werden. Diese Umsetzung hat jedoch Nachteile bei Anfragen, die alle Elemente einer Oberklasse betreffen, da hier zusätzlich manuell in den Unterklassen gesucht werden muss. Sie ist beispielsweise hier in allen Relationen zu suchen, wenn der Name des Mitarbeiters mit der Personalnummer 1278123 gesucht wird.

3.4.3.2 Partitionierungs-Modell – Vertikale Abbildung

Bei diesem Modell werden die Attribute analog zur Darstellung im ER-Diagramm aufgenommen, jede Relation erhält nur die auf der Vererbungsebene neu hinzugekommenen Attribute. Damit jedoch eine Verknüpfung der Einträge möglich ist, wird der Primärschlüssel in alle Relationen übernommen.

- ▶ Mitarbeiter (PNR, Name)
- ▶ Manager (PNR, Bereich)
- ▶ Entwickler (PNR, LetzteSchulung)
- ▶ Java-Entwickler (PNR, EU)
- ▶ C++-Entwickler (PNR, BS)

Die Tupel werden hier partitioniert. Jede Instanz wird entsprechend der Klassenattribute in der IS-A-Hierarchie zerlegt und in Teilen in den zugehörigen Klassen gespeichert. Es wird nur das Schlüssel-Attribut dupliziert. So wird beispielsweise bei einem Java-Entwickler die Entwicklungsumgebung in der Relation **Java-Entwickler**, der Termin der letzten Schulung in der Relation **Entwickler** und der Name in der Relation **Mitarbeiter** gespeichert.

Die Speicherungskosten sind aufgrund der Mehrfachspeicherung des Primärschlüssels geringfügig erhöht. Die Suche des Namens des Mitarbeiters mit der Personalnummer 1278123 kann hier jedoch wesentlich einfacher durchgeführt werden, da nur in einer Relation nachgeschaut werden muss. Möchte man aber alle Informationen eines Technikers, so sind diese aus verschiedenen Relationen zusammenzusetzen.

3.4.3.3 Volle Redundanz

Auch hier wird die Vererbung manuell durchgeführt, die Relationen sehen aus wie im Hausklassenmodell.

- ▶ Mitarbeiter (PNR, Name)
- ▶ Manager (PNR, Name, Bereich)
- ▶ Entwickler (PNR, Name, LetzteSchulung)
- ▶ Java-Entwickler (PNR, Name, LetzteSchulung, EU)
- ▶ C++-Entwickler (PNR, Name, LetzteSchulung, BS)

Jede Instanz wird wiederholt in jeder Klasse, der sie angehört, gespeichert. Sie besitzt dabei die Attributwerte, die geerbt wurden zusammen mit den Werten der Unterklasse. Dadurch können die Nachteile, die bei Anfragen bei den beiden zuvor vorgestellten Ansätzen auftraten, beseitigt werden. Nachteilig sind jedoch die erhöhten Speicherungskosten, sowie die Möglichkeit auftretender Änderungsanomalien. Ändert sich der Termin der letzten Schulung eines Java-Entwicklers, so muss dieser an zwei Stellen angepasst werden. Wird ein Wert vergessen, ist der Datenbestand inkonsistent.

3.4.3.4 Überrelation – Filtered Mapping

Der letzte Vorschlag erzeugt für die Hierarchie eine einzige Relation, die alle Attribute enthält, sowie ein Typ-Attribut.

- Mitarbeiter (PNR, Typ, Name, Bereich, LetzteSchulung, EU, BS)

Alle Tupel werden in dieser einen Relation abgelegt, indem die passenden Attribute belegt werden. Zur Erkennung des eigentlichen Entity-Typs wird ein zusätzliches Typ-Attribut aufgenommen. Attribute, die nicht zum Typ eines Tupels gehören, werden mit NULL belegt.

Die Nachteile der zuvor vorgestellten Lösungen sind alle behoben. Diese Lösung wird bei großen Hierarchien mit vielen Attributen jedoch schnell sehr unübersichtlich. Insbesondere muss der Benutzer wissen, welche Attribute zu welchem Typ gehören, wofür eine entsprechende Dokumentation immer bereit liegen muss. Das später eingeführte View-Konzept relationaler DBVS kann hier (und auch bei den anderen Ansätzen) helfen.

Kapitel 4

Relationale Entwurfstheorie

In den vorangegangenen Kapiteln wurde über eine Anforderungsanalyse ein konzeptioneller Entity-Relationship-Entwurf und schließlich ein relationales Schema erstellt. Dieses relationale Schema soll nun einer Feinabstimmung mit Hilfe formaler Methoden unterzogen werden, um potenzielle Probleme aufgrund von Redundanzen zu vermeiden.

Die Basis für diesen Feinentwurf bilden funktionale Abhängigkeiten, die eine Verallgemeinerung des Schlüsselbegriffs darstellen. Basierend auf diesen Abhängigkeiten werden Normalformen für Relationenschemata definiert. Die Normalformen dienen dazu, die 'Güte' eines Relationenschemas zu bewerten. Wenn für ein Relationenschema diese Normalformen nicht erfüllt sind, kann man es durch Anwendung entsprechender Normalisierungsalgorithmen in mehrere Schemata zerlegen, die dann die entsprechende Normalform erfüllen. Dadurch entstehen Entwürfe, die eine leichte Handhabung erlauben und gleichzeitig die Konsistenzherhaltung in der Datenbank vereinfachen.

4.1 Merkmale schlechter Relationenschemata

Schlecht entworfene Relationenschemata können zu so genannten Anomalien führen, die im folgenden anhand einer Lieferungsrelation beschrieben werden:

- Lieferung(Lieferant, Teil, Lieferdatum, Wohnort, Anzahl)

Einfügeanomalie: Bei diesem schlechten Entwurf wurden Informationen zweier Entity-Typen aus der realen Anwendungswelt vermischt, Lieferanten und Lieferungen. Deshalb treten Probleme auf, wenn man Informationen eintragen möchte, die nur zu einem Entity-Typ gehören. Will man beispielsweise Daten eines neuen Lieferanten eintragen, der jedoch noch keine Lieferung durchgeführt hat, so ist dies nicht möglich, da die Attribute Teil und Lieferdatum unbekannt sind und NULL-Werte beim Primärschlüssel nicht zulässig sind. Erst mit der ersten Lieferung ist der Lieferant eintragbar.

Löschanomalie: Wenn die Information bezüglich eines der zwei miteinander vermischten Entity-Typen gelöscht wird, kann es zum gleichzeitigen und unbeabsichtigten Verlust der Daten des anderen Entity-Typs kommen. Wird hier beispielsweise die letzte Lieferung eines Lieferanten gelöscht, so gehen auch die allgemeinen Informationen zu einem Lieferanten, wie seine Adresse, verloren.

Updateanomalie: Wenn ein Lieferant umzieht, muss dies in der Datenbank angepasst werden. Aufgrund des schlechten Schemas existiert die Information jedoch mehrfach, also redundant. Deshalb kann leicht der Fall eintreten, dass bei der Änderung Einträge übersehen werden.

Um einen solchen unzulänglichen Entwurf zu revidieren, werden bei der so genannten Normalisierung Relationenschemata aufgespalten. Dabei gibt es zwei grundlegende Korrektheitskriterien für eine solche Zerlegung von Relationenschemata:

Verlustlosigkeit: Die in der ursprünglichen Relationenausprägung enthaltenen Informationen müssen aus dem zerlegten Schema rekonstruierbar sein.

Abhängigkeitserhaltung: Informationen über die Abhängigkeiten zwischen Attributen müssen erhalten bleiben.

Für das Beispiel sehen die normalisierten Relationen wie folgt aus:

- ▶ Lieferant (LNR, Wohnort)
- ▶ Lieferung (LNR, Teil, Lieferdatum, Anzahl)

4.2 Funktionale Abhängigkeiten

Eine funktionale Abhängigkeit (functional dependency, FA) stellt eine Bedingung an die möglichen gültigen Ausprägungen des Datenbankschemas dar.

Definition 4.1 Seien X und Y Teilmenge der Attribute eines Relationenschemas R . Die funktionale Abhängigkeit $X \rightarrow Y$ (X bestimmt Y funktional) besteht, wenn für alle Tupel der Relation zu R gilt:

Zwei Tupel, deren Komponenten in X übereinstimmen, stimmen auch in Y überein.

$$\forall u \in R \forall v \in R (u[X] = v[X]) \rightarrow (u[Y] = v[Y])$$

Alternativ mit der später eingeführten Relationenalgebra: Die Relation R erfüllt die FA $X \rightarrow Y$, wenn für jeden X -Wert x der Ausdruck $\pi_Y(\sigma_{X=x}(R))$ höchstens ein Tupel hat.

Abbildung 4.1 zeigt eine grafische Darstellung zweier funktionaler Abhängigkeiten.



Abbildung 4.1: Grafische Darstellung funktionaler Abhängigkeiten

Zur Verdeutlichung dieses für die relationale Entwurfstheorie zentralen Konzepts der funktionalen Abhängigkeit betrachte man folgende Relation R :

A	B	C	D
a_4	b_2	c_4	d_3
a_1	b_1	c_1	d_1
a_1	b_1	c_1	d_2
a_2	b_2	c_3	d_2
a_3	b_2	c_4	d_3

Diese Relation erfüllt die FA $\{A\} \rightarrow \{B\}$, da es nur zwei Tupel mit gleichem A-Attributwert gibt. Bei diesen beiden Tupeln stimmt auch der Wert des Attributs B überein. Weiterhin erfüllt die gezeigte Ausprägung die FA $\{A\} \rightarrow \{C\}$. Außerdem ist die FA $\{C, D\} \rightarrow \{B\}$ in der Relation erfüllt. Die FA $\{B\} \rightarrow \{C\}$ ist in der Relation nicht erfüllt, da die letzten beiden Zeilen bei gleichem B-Wert unterschiedliche C-Werte haben.

Es soll an dieser Stelle nochmals betont werden, dass funktionale Abhängigkeiten eine semantische Konsistenzbedingung darstellen, die sich aus der jeweiligen Anwendungssemantik und nicht aus der derzeitigen zufälligen Relationenausprägung ergibt. Funktionale Abhängigkeiten müssen zu allen Zeiten und jedem Datenbankzustand eingehalten werden. Es ist Aufgabe des Datenbankentwerfers die funktionalen Abhängigkeiten aus der Anwendungssemantik zu bestimmen.

Folgende FAs gelten immer:

- ▶ triviale FA: $X \rightarrow X$
- ▶ Falls X Schlüsselkandidat eines Relationenschemas R ist, dann gilt für alle Y aus R: $X \rightarrow Y$

Definition 4.2 Es gelte $\{A_1, A_2, \dots, A_n\} \rightarrow \{B_1, B_2, \dots, B_m\}$.

$B = \{B_1, B_2, \dots, B_m\}$ ist voll funktional abhängig von $A = \{A_1, A_2, \dots, A_n\}$, wenn B funktional abhängig von A ist, aber nicht funktional abhängig von einer echten Teilmenge von A ist.

$A \rightarrow B$ ist eine partielle Abhängigkeit, wenn ein Attribut A_i in A existiert, so dass $(A - A_i) \rightarrow B$ gilt.

Ein einfaches Beispiel für eine partielle FA ist $\{\text{PNR}, \text{Gehalt}\} \rightarrow \{\text{Name}\}$, da die PNR alleine ausreicht den Namen zu bestimmen.

4.3 Normalformen

Der Normalisierungsprozess untersucht Relationenschemata anhand formaler Kriterien und zerlegt diese bei Bedarf um dadurch potenzielle Anomalien zu vermeiden. Ausgangspunkt für die Darstellung soll folgende unnormalierte Relation 'Prüfung' sein:

VL_ID	VL_Name	Semester	STUDENT
EinWI	Einführung WI	1	(44767, Maier, ...) (53264, Schulze, ...) ...
PM	Prozessmodellierung	5	(82319, Lohner, ...) ...

Diese Relation enthält Attribute, die wiederum Relationen sind, wodurch sich folgende Nachteile ergeben:

- ▶ Unsymmetrie (nur eine Richtung der Beziehung)
- ▶ Redundanzen (bei n:m-Beziehungen), Änderungsanomalien
- ▶ Implizite Darstellung von Informationen

4.3.1 Erste Normalform

Die erste Normalform verlangt, dass alle Attribute atomare Wertebereiche haben. Diese Voraussetzung ist Bestandteil des relationalen Datenmodells, zusammengesetzte, mengenwertige oder gar relationenwertige Attributwertebereiche sind nicht zulässig.¹

¹Es gibt Entwicklungen im Datenbankbereich, in denen gerade auf die Einhaltung der ersten Normalform verzichtet wird. Dies Modell wird auch als NF^2 -Modell (non-first-normal-form Modell) bezeichnet.

Da die gegebene Darstellung nicht in erster Normaform ist, muss sie als erstes flachgeklopft werden. Dazu wird die untergeordneten Relationen aus der Vaterrelation entfernt. Jede untergeordnete Relation wird um den Primärschlüssel der übergeordneten Relation zu einer selbstständigen Relation erweitert. Dieser Prozess ist bei Bedarf rekursiv zu wiederholen.

In diesem Beispiel ergeben sich dadurch folgende zwei Relationen:

- ▶ Vorlesung (VL_ID, VL_Name, Semester)
- ▶ Prüfung (VL_ID, MATRNR, Name, GebDat, ADR, Kurs, Studienrichtung, SGName, SGLeiter, PDAT, Note)

4.3.2 Zweite Normalform

Die erste Normalform verursacht noch immer viele Änderungsanomalien, da verschiedene Objekt-Mengen in einer Relation gespeichert werden können. Hierdurch können insbesondere Einfüge-, Lösch- und, aufgrund von Redundanzen, Änderungsanomalien auftreten. Die zweite Normalform vermeidet einige der Anomalien dadurch, dass nicht voll funktional (partiell) abhängige Attribute eliminiert werden. Dadurch wird eine Separierung verschiedener Objektarten in eigene Relationen erreicht.

Definition 4.3 Ein Primärattribut (Schlüsselattribut) eines Relationenschemas ist ein Attribut, das zu mindestens einem Schlüsselkandidaten des Schemas gehört.

Definition 4.4 Ein Relationenschema R ist in 2NF, wenn es in 1NF ist und jedes Nicht-Primärattribut von R voll funktional von jedem Schlüsselkandidaten in R abhängt.

Einzigster Schlüsselkandidat der Prüfungsrelation ist die Kombination VL_ID und MATNR. Betrachtet man die voll funktionalen Abhängigkeiten der Studentenrelation (Abbildung 4.2), so erkennt man, dass nur PDAT und NOTE von diesem Schlüsselkandidaten voll funktional abhängig sind, die anderen Attribute sind von der MATNR abhängig und damit nur partiell funktional abhängig von dem Schlüsselkandidaten. Damit muss wie folgt normalisiert werden:

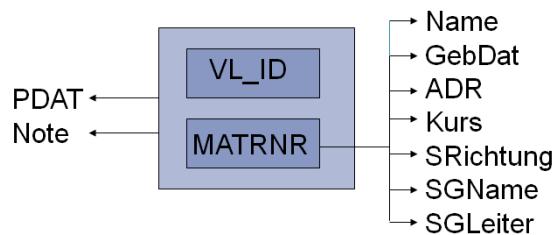


Abbildung 4.2: Voll funktionale Abhängigkeiten der Studentenrelation

1. Bestimme funktionale Abhängigkeiten zwischen Nicht-Primärattributen und Schlüsselkandidaten
2. Eliminiere partiell abhängige Attribute und fasse sie in einer eigenen Relation unter Hinzunahme der zugehörigen Primärattribute zusammen.

Durch diese Normalisierung ergeben sich die in Abbildung 4.3 dargestellten Relationen.

Man erkennt in dieser Darstellung jedoch schnell, dass einige Informationen noch redundant gespeichert werden und Änderungsanomalien noch auftreten können. Auch dies gilt es zu vermeiden.

Prüfung

VL_ID	MATRNR	PDAT	NOTE
PM	44767	20.9.2020	2.3
PM	53264	21.9.2021	1.7
EinWI	82319	21.9.2021	1.7

Vorlesung

VL_ID	VL_Name	Semester
EinWI	Einführung WI	1
PM	Prozessmodellierung	5

Student

MATRNR	Name	ADR	GebDat	Kurs	SGName	SRic	SGL
44767	Maier	...	21.05.01	WIDS20..	Wirtsch...	Data Sc.	Daurer
53264	Schulze	...	12.12.05	WIDS20..	Wirtsch...	Data Sc.	Daurer
82319	Lohner	...	16.03.04	WIDS20..	Wirtsch...	Data Sc.	Daurer

Abbildung 4.3: Beispiel in zweiter Normalform

4.3.3 Dritte Normalform

Die in Abbildung 4.3 erkennbaren Redundanzen entstehen aufgrund transitiver Abhängigkeiten, die Studentenrelation enthält sowohl Studenten als auch daraus resultierende Studiengangsdaten.

Definition 4.5 Eine Attributmenge Z von Relationenschema R ist transitiv abhängig von einer Attributmenge X in R , wenn gilt:

- X und Z sind disjunkt
- Es existiert eine Attributmenge Y in R , so dass gilt:

$$X \rightarrow Y, Y \rightarrow Z, \text{nicht } Y \rightarrow X, Z \not\subseteq Y$$

Die Einschränkung 'nicht $Y \rightarrow X$ ' ist wichtig, da hierdurch verhindert wird, dass Alternative Schluesselkandidaten zu transitiven Abhängigkeiten führen, die eliminiert werden müssen, da für zwei Schluesselkandidaten S_1 und S_2 immer $S_1 \rightarrow S_2 \rightarrow X$ gilt. Diese Abhängigkeit ist jedoch nicht störend.

Betrachtet man die funktionalen Abhängigkeiten der Studentenrelation (ausschnittsweise in Abbildung 4.4), erkennt man, dass SGNAME und SG transitiv von der Matrikelnummer abhängig sind. Auch SG ist über den Weg MATNR → SGNAME → SG transitiv abhängig von MATNR.

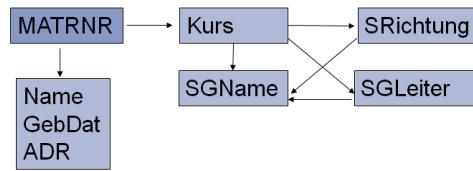


Abbildung 4.4: Transitive Abhängigkeiten in der Studentenrelation

Definition 4.6 Ein Relationenschema R befindet sich in 3NF, wenn es sich in 2NF befindet und jedes Nicht-Primär-Attribut von R von keinem Schluesselkandidaten von R transitiv abhängig ist.

Damit müssen für die dritte Normalform alle Nicht-Primär-Attribute, die transitiv von einem Schlüsselkandidaten abhängig sind, aus der Relation entfernt werden und in einer eigenen Relation abgelegt werden. Damit jedoch weiterhin eine Zuordnung der Information möglich ist, wird die Attributmenge, die bei der Transitivitätsanalyse in der Mitte steht, mit in die neue Relation aufgenommen und dort als Primärschlüssel verwendet. Für unser Studentenbeispiel ergeben sich damit die in Abbildung 4.5 dargestellten Relationen in dritter Normalform, wobei transitive Abhängigkeiten hier auch in einer Auslagerungstabelle entstanden sind, die dadurch weiter zerlegt werden muss.

Prüfung				Student				
VL_ID	MATRNR	PDAT	NOTE	MATRNR	Name	ADR	GebDat	Kurs
PM	44767	20.9.2020	2.3	44767	Maier	...	21.05.2001	WWIDS20..
PM	53264	21.9.2021	1.7	53264	Schulze	...	12.12.2005	WWIDS20..
EinWI	82319	21.9.2021	1.7	82319	Lohner	...	16.03.2004	WWIDS20..

Vorlesung			Kurse		
VL_ID	VL_Name	Semester	Kurs	Studienrichtung	SGLeiter
EinWI	Einführung WI	1	WWIDS20..	Data Science	Daurer
PM	Prozessmodellierung	5	WWIBE120..	Business Engineering	Lehmann
			WWIBE220..	Business Engineering	Lehmann

Studienrichtung	
Richtung	SGName
Data Science	Wirtschaftsinformatik
Business Engineering	Wirtschaftsinformatik

Abbildung 4.5: Beispiel in dritter Normalform

4.3.4 Boyce-Codd-Normalform

Die Definition der 3NF hat Schwächen bei Relationen mit mehreren Schlüsselkandidaten, wenn die Schlüsselkandidaten zusammengesetzt sind und sich überlappen. Dabei kann es vorkommen, dass ein Teil eines zusammengesetzten Schlüsselkandidaten funktional abhängig ist von einem Teil eines anderen Schlüsselkandidaten. Daher wurde die 3NF zur Boyce-Codd-Normalform weiterentwickelt. Man betrachte hierzu folgende Beispieldrelation:

- ▶ PROJEKTMITARBEIT (PERSNR, PROJNAME, TELNR, AUFGABE)

Zwischen PERSNR und TELNR besteht hier eine 1:1-Beziehung, so dass auch die Kombination (PROJNAME, TELNR) ein Schlüsselkandidat der Relation ist. Damit ist AUFGABE das einzige nicht-Primärattribut, die Relation befindet sich in dritter Normalform. Trotzdem kann diese Relation Redundanzen enthalten. Ist beispielsweise ein Mitarbeiter an mehreren Projekten beteiligt, so wird seine Telefonnummer bei jeder Projektmitarbeit redundant geführt, Updateanomalien sind möglich. Ziel der Boyce-Codd-Normalform ist die Beseitigung dieser Anomalie für Primärattribute.

Definition 4.7 Ein Attribut (oder eine Attributgruppe), von denen andere voll funktional abhängen, heißt Determinant.

Die Determinanten in der Relation Projektmitarbeit sind:

- ▶ PERSNR, PROJNAME (da \rightarrow AUFGABE)
- ▶ TELNR, PROJNAME (da \rightarrow AUFGABE)
- ▶ PERSNR (da \rightarrow TELNR)
- ▶ TELNR (da \rightarrow PERSNR)

Definition 4.8 Ein Relationenschema R ist in BCNF, wenn jeder Determinant ein Schlüsselkandidat von R ist.

In dem Beispiel ist dies nicht gegeben, da PERSNR und TELNR jeweils Determinanten sind, jedoch keiner von beiden ein Schlüsselkandidat ist. Um die BCNF zu erreichen, müssen alle Attribute entfernt werden, die dies verursachen. Entfernt man beispielsweise die TELNR aus der Relation, so ist die Bedingung gegeben. Auch das Entfernen der PERSNR erfüllt die Ansprüche. Hier hat man also die Wahl des zu entfernenden Attributs. Natürlich muss dies weiterhin in der Datenbank bestehen bleiben, eine neue Relation mit Bezug auf die Ausgangsrelation ist hier anzulegen, wobei die geschilderten zwei Möglichkeiten vorliegen:

- ▶ PROJEKTMITARBEIT (PERSNR, PROJNAME, AUFGABE)
PERSONAL (PERSNR, TELNR)
- ▶ PROJEKTMITARBEIT (TELNR, PROJNAME, AUFGABE)
PERSONAL (PERSNR, TELNR)

Beide Zerlegungen führen auf BCNF-Relationen, wobei die Änderungsanomalie eliminiert wurde. Alle funktionalen Abhängigkeiten bleiben bei der Aufspaltung erhalten.

Die Boyce-Codd-Normalform stimmt mit der dritten Normalform überein, wenn eine der folgenden Bedingungen erfüllt ist:

- ▶ Es existiert nur ein Schlüsselkandidat.
- ▶ Es existieren nur einfache Schlüsselkandidaten (1 Attribut).
- ▶ Die Schlüsselkandidaten überlappen sich nicht.

Die Boyce-Codd-Normalform (häufig auch die dritte Normalform) wird häufig als minimale Zielsetzung bei der Normalisierung von Relationen betrachtet.

4.3.5 Vierte Normalform

Die vierte Normalform untersucht eine Verallgemeinerung der funktionalen Abhängigkeiten, die mehrwertigen Abhängigkeiten. Durch eine FA wird jeweils höchstens ein Wert des abhängigen Attributs bestimmt. Mehrwertige Abhängigkeiten (MWA) sind Generalisierungen von funktionalen Abhängigkeiten, sie bestimmen jeweils eine Menge von Werten des abhängigen Attributs. MWA entstehen durch zwei oder mehr unabhängige Attribute im Schlüssel einer Relation.

Man betrachte beispielsweise folgende Relation mit Informationen zu den Vornamen von Kindern von Mitarbeitern, bei der der Primärschlüssel aus beiden Attributen zusammengesetzt ist: KinderDerMitarbeiter(PNR, Vorname). Hier liegt eine Mehrwertige Abhängigkeit vor: $\text{PNR} \rightarrow\rightarrow \text{Vorname}$.

Definition 4.9 X und Y seien Attributmengen des Relationenschemas R . Die mehrwertige Abhängigkeit MWA $X \rightarrow\rightarrow Y$ gilt in R genau dann, wenn die Menge der Y -Werte, die zu einem $(X\text{-Wert}, Y\text{-Wert})$ -Paar gehören, nur vom X -Wert bestimmt sind.

Eine mehrwertige Abhängigkeit einer Attributmengen Y von einer Attributmengen X ist trivial, wenn Y Teil von X ist oder die Relation nur aus X und Y besteht.

Bei vorherigem Beispiel zu den Vornamen von Mitarbeitern handelt es sich damit um eine triviale mehrwertige Abhängigkeit. Probleme entstehen aber, wenn mehrere mehrwertige Abhängigkeiten in einer Relation auftreten. Im Beispiel:

<u>PNR</u>	Fähigkeit	<u>VenameKind</u>
123	Englisch	Hannah
123	Englisch	Max
123	Prog.	Hannah
123	Prog.	Max



Diese Relation ist in BCNF, jedoch sind Informationen redundant oder man muss mit NULL-Werten arbeiten, was beim Primärschlüssel jedoch nicht möglich ist.

Definition 4.10 X, Y und Z seien Attributmengen des Relationenschemas R . Ist sowohl Y als auch Z mehrwertig abhängig von X , so schreibt man $X \rightarrow\rightarrow Y \mid Z$.

Eine Relation ist in vierter Normalform, wenn sie in Boyce-Codd-Normalform ist und für jede mehrwertige Abhängigkeit einer Attributmenge Y von einer Attributmenge X gilt:

- ▶ Die mehrwertige Abhängigkeit ist trivial oder
- ▶ X ist ein Schlüsselkandidat der Relation.

Liegt damit eine mehrwertige Abhängigkeit der Form $X \rightarrow\rightarrow Y \mid Z$ vor, ist die vierte Normalform nicht gegeben, da die mehrwertige Abhängigkeit nicht trivial ist und X alleine kein Schlüsselkandidat ist.

Folglich muss diese Relation zerlegt werden:

- ▶ R1 (PNR, FÄHIGKEIT) und
- ▶ R2 (PNR, KIND)

Zur Überführung einer Relation in die vierte Normalform sind Relationenschemata mit mehrwertigen Abhängigkeiten der Art $X \rightarrow\rightarrow Y \mid Z$ in zwei neue Relationenschemata mit den Attributen X,Y bzw. X,Z zu zerlegen.

4.3.6 Fünfte Normalform

Man betrachte folgendes Beispiel:

<u>PNR</u>	Projekt	Fähigkeit
123	P1	Englisch
123	P2	Englisch
123	P2	Programmieren

Diese Relation ist in vierter Normalform. Insbesondere würde eine Zerlegung in zwei Projektionen R1 (PNR, PROJEKT) und R2(PNR, FÄHIGKEIT) zum Verlust von Informationen führen, da ein JOIN auf den Projektionen vorher nicht existente Tupel erzeugt (connection trap).

Eine Zerlegung der Beispielrelation R in drei Projektionen R1(PNR, PROJEKT), R2(PNR, FÄHIGKEIT) sowie R3(PROJEKT, FÄHIGKEIT) ist verlustfrei, d.h. ein Join zwischen diesen drei Relationen erzeugt genau die Ausgangsrelation.

Definition 4.11 Zerlegbarkeits-Constraints ($n=3$): R habe die Schlüsselteile A, P und F. Die verlustfreie Zerlegbarkeit in $R1(A,P)$, $R2(P,F)$, $R3(F,A)$ kann durch folgenden Constraint formal ausgedrückt werden:

IF $(a1,p1,f2), (a2,p1,f1), (a1,p2,f1)$ in R THEN $(a1,p1,f1)$ in R

Der 3-decomposable Constraint ist genau dann erfüllt, wenn die Relation stets gleich dem Join der Projektionen ist. Dieser Constraint heißt deshalb Join-Abhängigkeit.

Definition 4.12 Ein Relationenschema R erfüllt die Join-Abhängigkeit

$*(X, Y, \dots, Z)$ gdw. für alle Relationen R aus R gilt:

$$R = \pi_X(R) \bowtie \pi_Y(R) \bowtie \dots \bowtie \pi_T(R)$$

Mit der Join-Abhängigkeit lässt sich eine bestimmte Anwendungssemantik für eine Relation R ausdrücken, die bei allen Modifikationsoperationen gewahrt werden muss. In der Relation R (PNR, PROJEKT, FÄHIGKEIT) besteht die Join-Abhängigkeit $*(R1, R2, R3)$ mit $R1(PNR, PROJEKT)$, $R2(PNR, FÄHIGKEIT)$ sowie $R3(PROJEKT, FÄHIGKEIT)$. Die dadurch festgelegte Anwendungssemantik lässt sich umgangssprachlich wie folgt beschreiben: Jeder Angestellte A mit Fähigkeit F, der in Projekt P mitarbeitet, welches Fähigkeit F verlangt, muss seine Fähigkeit in P auch einsetzen.

Um die Join-Abhängigkeit auf R zu erhalten, sind bei Aktualisierungsoperationen des Benutzers zusätzliche Modifikationsoperationen auf R erforderlich, die sehr undurchsichtig sind. Es treten also Anomalien auf. Ziel ist deshalb eine verlustfreie Zerlegung einer Relation in drei (n) Projektionen, so dass diese Anomalien nicht auftreten.

Definition 4.13 Ein Relationenschema R ist in 5NF genau dann, wenn jede Join-Abhängigkeit in R auf einen Schlüsselkandidaten von R zurückgeht.

- ▶ 5NF behandelt N:M-Beziehungen zwischen drei und mehr Schlüsselattributen.
- ▶ 5NF ist frei von Anomalien, die durch Projektion eliminiert werden können.

4.3.7 Zusammenfassung

- ▶ 1NF: Ein Relationenschema R ist in 1NF, wenn alle seine Wertebereiche nur atomare Werte besitzen.
- ▶ 2NF: Ein Relationenschema R ist in 2NF, wenn es in 1NF ist und jedes Nicht-Primärattribut von R voll funktional von jedem Schlüsselkandidaten in R abhängt.
- ▶ 3NF: Ein Relationenschema R befindet sich in 3NF, wenn es sich in 2NF befindet und jedes Nicht-Primär-Attribut von R von keinem Schlüsselkandidaten von R transitiv abhängig ist.
- ▶ BCNF: Ein Relationenschema R ist in BCNF, wenn jeder Determinant ein Schlüsselkandidat von R ist.
- ▶ 4NF: Ein Relationenschema R ist in 4NF, wenn es in BCNF ist und jede MWA trivial ist oder von einem Schlüsselkandidaten ausgeht.
- ▶ 5NF: Ein Relationenschema R ist in 5NF genau dann, wenn jede Join-Abhängigkeit in R auf einen Schlüsselkandidaten von R zurückgeht.

Die Herstellung der 5NF ist nicht immer notwendig oder sinnvoll. Betrachtet man die (Teil-)Relation PERS (PNR, PLZ, Straße, Vorwahl,...) mit der funktionalen Abhängigkeit (PLZ, Straße) → Vorwahl, so ist diese nicht in dritter Normalform, da diese eine Trennung der Vorwahl verlangt.

Damit können Änderungsanomalien auftreten. Doch wie häufig ändert sich die Vorwahl zu einer Adresse? Hier kann man bewusst die Relation in zweiter Normalform belassen.

Es gibt auch Situationen, bei denen man bewusst auf die Normalisierung gemäß Boyce-Codd-Normalform verzichtet:

Beispiel:

⇒ SFP (Student, Fach, Prüfer)

Hierbei gelte, dass jeder Prüfer nur ein Fach prüft, aber ein Fach von mehreren Prüfern geprüft werden kann. Weiterhin legt jeder Student in einem Fach nur eine Prüfung ab. Damit gilt: {Student, Fach} → {Prüfer}, {Prüfer} → {Fach}

Da in diesem Beispiel {Student, Prüfer} auch ein Schlüsselkandidat ist, müsst man für die Boyce-Codd-Normalform die Relation zerlegen:

- ▶ SP (Student, Prüfer)
- ▶ PF (Prüfer, Fach)

Durch die Zerlegung ist nun die Beziehung zum Fach ausgelagert. Da aber gerade das interessanter ist, wie der Zusammenhang eines Studenten zu seinem Prüfer, kann man hier auf die Normalisierung verzichten. Insbesondere geht durch die Normalisierung die FA $\{Student, Fach\} \rightarrow \{Prüfer\}$ verloren, die BCNF ist hier zu streng, um bei der Zerlegung alle funktionalen Abhängigkeiten zu bewahren.

Übung 4.1 Gegeben sei eine Tabelle Flugbuchung mit folgenden Attributen:

- ▶ PassagierID (Dauerhaft einem Passagier zugeordnet)
- ▶ FlugzeugKZ
- ▶ FlugZeitpunkt (Datum und Uhrzeit)
- ▶ Passagiername
- ▶ KürzelFluggesellschaft
- ▶ SitzplätzeImFlugzeug
- ▶ NameFluggesellschaft
- ▶ SitzDerFluggesellschaft
- ▶ Flugzeugherrsteller (z.B. Boeing, Airbus)
- ▶ FlugzeugTyp (z.B. 747, A300)
- ▶ Durchschnittliche Reichweite
- ▶ Spannweite
- ▶ Startflughafen
- ▶ Zielflughafen
- ▶ ExtraHeavy (ja oder nein)
- ▶ Vegetarier (ja oder nein)

- ▶ *Tankvolumen*
- ▶ *Durchschnittlicher Treibstoffverbrauch*
- ▶ *Sitzplatz (des Passagiers beim Flug)*

Machen Sie Vorschläge für funktionale Abhängigkeiten in dieser Tabelle und Normalisieren sie die Tabelle schrittweise entlang der Normalisierungsfolge in die 4. Normalform.

Kapitel 5

Die Sprache SQL

Das relationale Datenbankmodell von Dr. E. F. Codd aus dem Jahr 1970 ist eine theoretische Beschreibung dessen, wie relationale Datenbanken entworfen werden, und nicht darüber, wie sie zu benutzen sind. Um Relationen mit den dazugehörigen Attributen zu erstellen, um Beziehungen zwischen Relationen aufzubauen oder die Daten in der Datenbank zu manipulieren, bedarf es jedoch einer einfach einsetzbaren Datenbank-Programmiersprache.

Im Rahmen der Entwicklung des ersten relationalen Datenbanksystems 'System R' bei IBM wurde auch eine Datenbanksprache namens SQL (Structured Query Language) definiert und integriert. Entwicklungsziel war dabei eine relational vollständige Sprache, die damit gleichmäßig zur Relationalen Algebra ist und die dabei leicht zu erlernen ist. Zur Überprüfung dieser Zielsetzung wurden beispielsweise Untersuchungen zur Erlernbarkeit in Schulen und Universitäten durchgeführt.

Besondere Gewichtung erhielt die Sprache dann auch durch die ersten DBMS-Produkte, insbesondere durch das DBMS Oracle, welches sei 1979 kommerziell für DEC-Minicomputer verfügbar war. So konnte sich die Sprache schnell als Quasi-Standard für DBMSs etablierte. Seit Ende der 70er Jahre hat sich SQL in vielfältiger Weise weiterentwickelt. Zum einen wurde früh erkannt, dass nur eine herstellerunabhängige Weiterentwicklung der Sprache die notwendige Offenheit und Kompatibilität zwischen den einzelnen Produkten sicherstellen würde. Sie wurde durch das American National Standard Institute (ANSI) über die letzten knapp 30 Jahre standardisiert und kontinuierlich erweitert. 1986 wurde mit SQL1 der erste Sprachstandard von ANSI verabschiedet, dem eine weitere Version unter dem Namen SQL2 bzw. SQL-92 mit wichtigen Erweiterungen folgte. Erheblich verändert wurde die Sprache mit dem 1999 verabschiedeten SQL3-Standard, dem bisher letzten Standard, dessen Dokumentumfang fast 1700 Seiten umfasst, etwa der dreifache Umfang des ursprünglichen Standards SQL1 (600 Seiten). Insbesondere die Erweiterung um objektrelationale Eigenschaften in den 90er Jahren führte zu objektrelationalen Erweiterungen. Entwicklungen im Kontext der Sprache XML kamen um die Jahrtausendwende hinzu. 2004 folgte SQL:2003 bzw. SQL4 mit Window-Funktionen und Sequenzen. 2006 kamen weitere XML-Ergänzungen hinzu. SQL:2008 wurde um neue Triggerarten ergänzt, 2011 folgten Ergänzungen bei zeitbezogenen Funktionen. Als wichtige Alternative zu XML für den Datenaustausch hat mit SQL:2016 um Verarbeitungsfunktionen für JSON-Dokumente ergänzt. Mit SQL:2019 folgten Ergänzungen im Bereich mehrdimensionaler Felder.

SQL ist die De-facto-Standardsprache, mit der sich Daten aus relationalen Datenbanken abrufen und dort manipulieren lassen. Mit SQL kann der Programmierer oder Datenbankadministrator folgende Aufgaben ausführen:

- ▶ Die Struktur einer Datenbank definieren und modifizieren.
- ▶ Die Einstellungen der Systemsicherheit ändern.

- ▶ Benutzerberechtigungen für Datenbanken oder Tabellen einrichten.
- ▶ Eine Datenbank nach Informationen abfragen.
- ▶ Den Inhalt einer Datenbank aktualisieren.

In SQL unterscheidet man dabei zwischen mehreren Befehlsgruppen:

- ▶ Befehle, die zur Datendefinition dienen. Diese Befehlsgruppe wird als Data Definition Language (DDL) bezeichnet.
- ▶ Befehle, die zur Manipulation der Daten verwendet werden. Diese Gruppe wird Data Manipulation Language (DML) genannt. Der SELECT-Befehl zur Selektion von Datensätzen wird ebenfalls dieser Befehlsgruppe zugeordnet.
- ▶ Befehle zur Vergabe von Zugriffsrechten (DCL, Data Control Language).
- ▶ Befehle, die zur Transaktionsverarbeitung notwendig sind.

5.1 Demodatenbanken

Zur Veranschaulichung der Sprachkonstrukte werden Beispiele verwendet, die auf zwei Demodatenbankbereichen aufbauen. Dies ist zum einen eine kleine Personaldatenbank zum Abspeichern von Abteilungen und Personen. Eine Abteilung hat dabei eine eindeutige Nummer, einen Namen und ein Budget. Ein Mitarbeiter hat eine Personalnummer, einen Namen, ein Geburtsdatum, ein Gehalt und kann einer Abteilung sowie einem Vorgesetztem zugeordnet sein.

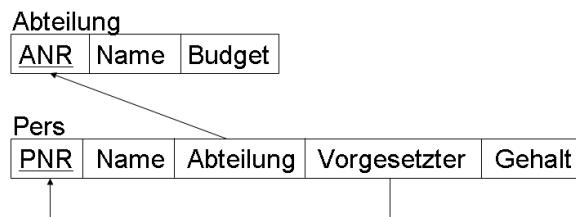


Abbildung 5.1: Die Personaldatenbank

Das zweite Beispiel ist etwas komplexer, es werden Grundelemente eines Flugbetriebs abgebildet. Dies beinhaltet Piloten, die u.a. einen nächsten Untersuchungstermin innerhalb der nächsten drei Monate haben. Piloten fliegen Flugzeuge, die eindeutige Kennzeichen haben und einer Fluggesellschaft zugeordnet sind.¹ Fluggesellschaften haben wiederum einen Basisflughafen an einem Ort mit einer Anzahl an Start- und Landebahnen.

5.2 Datendefinition

Bevor die Elemente einer Datenbank wie Tabellen und Zugriffsstrukturen angelegt werden können, müssen Festplattenzuordnungen und Speicherbereiche zugeordnet werden. Diese Aufgaben sind sehr produktpezifisch, so dass hier nicht näher darauf eingegangen werden kann.

¹Flüge werden mit Start- und Zielort angegeben. Dies ist eine stark vereinfachte Darstellung, die Angabe von einem Start- und Zielflughafen wäre hier in einer Anwendungsdatenbank sinnvoller.

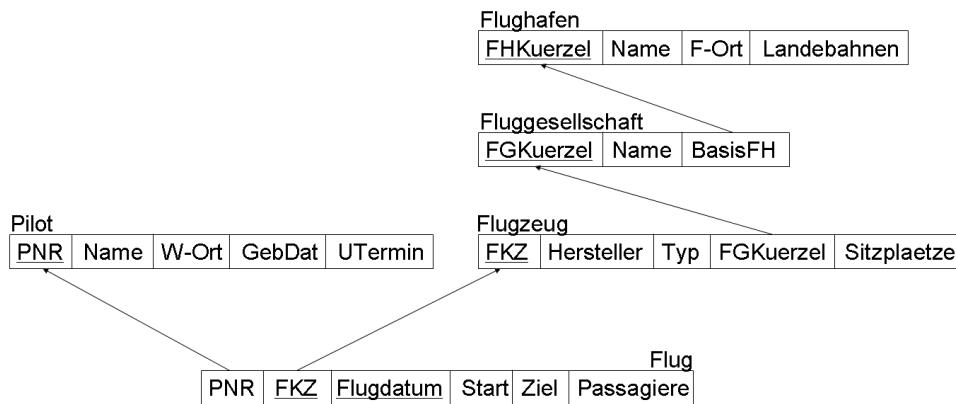


Abbildung 5.2: Die Flugbetriebdatenbank

5.2.1 Schemata

Schemata dienen zur benutzerspezifischen Gruppierung von Datenbankobjekten. Dies erlaubt es, dass Datenbankobjekte mit gleichem Namen, aber innerhalb verschiedener Schemata eingesetzt werden können. Insbesondere ist jedes Datenbankobjekt genau einem Schema zugeordnet.

```
/-----\
CREATE SCHEMA [schema] AUTHORIZATION user
[DEFAULT CHARACTER SET char-set]2
\-----/
```

Jedes Schema ist genau einem Benutzer zugeordnet. Falls kein expliziter Schemaname angegeben ist, erhält das Schema den Namen des Benutzers.

Beispiel:

```
⇒ CREATE SCHEMA FlugDB AUTHORIZATION LH_DBA1
```

Das Anlegen von Schemata wird nicht von allen Datenbanksystemen voll unterstützt. So kennt Oracle Schemata, diese haben jedoch immer den gleichen Namen wie der Benutzer. Sie werden automatisch beim Anlegen jedes Benutzers (CREATE USER) angelegt.

5.2.2 Datentypen

Das relationale Datenmodell sieht für jedes Attribut einer Relation die Angabe eines Datentyps vor. Im SQL-Standard stehen die in Tabelle 5.1 angegebenen Datentypen zur Verfügung:

5.2.3 Domänen

Der SQL2-Standard erlaubt, mittels dem CREATE DOMAIN-Befehl eigene Datentypen zu definieren. In derzeitigen SQL-Implementierungen ist er jedoch selten zu finden. Formal ist eine Domäne ein festgelegter Wertebereich für einen Standarddatentyp. So ist es nicht möglich, auf Basis einer Domäne eine neue Domäne zu definieren.

²Auf Groß-/Kleinschreibung bei den SQL-Befehlen ist nicht zu achten. In den Beispielen werden aus Gründen der Übersichtlichkeit für SQL-Schlüsselwörter Großbuchstaben verwendet.

CHARACTER [(n)]	Zeichenfolge fester Länge n, Default 1
CHAR [(n)]	
VARCHAR (n)	Zeichenfolge variable Länge und Maximallänge n
DECIMAL [(g,d)]	Zahl mit g (Stelligkeit) Gesamtstellen
DEC [(g,d)]	bei d Nachkommastellen,
NUMERIC [(g,d)]	Default d ist 0
NUMBER [(g,d)]	
REAL	Gleitkommazahl mit niedriger Genauigkeit
FLOAT	Gleitkommazahl mit mittlerer Genauigkeit
DOUBLE	Gleitkommazahl mit hoher Genauigkeit
INTEGER	Ganzzahl 32 Bit
INT	
BIGINT	Ganzzahl 64 Bit
SMALLINT	Ganzzahl 16 Bit
TINYINT	Ganzzahl 8 Bit
DATE	Datumswert
TIME	Zeitwert
TIMESTAMP	Datums- und Zeitwert

Tabelle 5.1: SQL-Standard-Datentypen

```
/ CREATE DOMAIN Domänenname [AS] Datentyp
  [ DEFAULT {Wert | USER | NULL} ]
  [ NOT NULL ]
  [ CONSTRAINT Regelname] CHECK (Bedingung)
\
```

Jede Domäne hat einen eindeutigen Namen und bezieht sich auf einen Grunddatentyp. Durch die darauf folgenden Angaben kann dann die Domäne weiter spezifiziert werden:

DEFAULT: Der hier hinterlegte WERT wird als Defaultwert verwendet. Verwendet eine Spalte diese Domäne und wird beim Einfügen eines Datensatzes kein Wert angegeben, wird der Defaultwert als Spaltenwert verwendet. Der Standard-Defaultwert ist NULL, dies kann hier aber auch explizit angegeben werden. Auch USER ist ein möglicher Defaultwert, der Wert entspricht dem Loginnamen des angemeldeten Benutzers.

NOT NULL: Mit dieser Option werden Domänen definiert, für die ein Wert obligatorisch ist, d.h. NULL ist kein zulässiger Wert.

CHECK: Hier kann eine Bedingung hinterlegt werden, die ein Wert bezüglich dieser Domäne erfüllen muss. Dies sind meist Bereichsbedingungen, siehe hierzu auch die Beispiele.

CONSTRAINT: Der bei CHECK angegebene Bedingung kann durch diese Option einen Namen gegeben werden. Dies erleichtert die Fehlersuche, wenn ein Wert aufgrund einer verletzten Bedingung nicht akzeptiert wird. In diesem Fall gibt das Datenbanksystem meist den Namen der Bedingung mit an, so dass man dann den Sachverhalt schnell überprüfen kann. Gibt man keinen Namen für die Bedingung an, vergibt das System einen internen Namen.

Beispiele:

```
⇒ CREATE DOMAIN DGehalt AS NUMERIC(10,2)
    NOT NULL
    DEFAULT 36000.00
```

Die Domäne DGehalt entspricht einer Zahl mit zwei Nachkommastellen. Die Angabe eines Werts ist Pflicht, das Standardgehalt und damit der Default liegt bei 36000 Euro.

```
⇒ CREATE DOMAIN DALter AS INT
    NOT NULL
    CHECK (VALUE > 16)
```

Altersangaben sind Ganzzahlen, die angegeben werden und dabei mindestens den Wert 16 annehmen müssen. So kann beispielsweise ein Mindestalter in einer Personaldatenbank definiert werden.³

5.2.4 Tabellen

Relationen sind die zentralen Elemente einer relationalen Datenbank. Zum Anlegen einer Relation dient der Befehl CREATE TABLE:

```
/—————\  
CREATE TABLE [Schema.]Tabellenname (  
    {<Spaltendefinition> | <Tabellenbedingung>}  
        [, <Spaltendefinition> | <Tabellenbedingung>] ...  
    )  
—————/
```

Jede Tabelle erhält einen Namen, der innerhalb des Schemas eindeutig sein muss. Tabellennamen müssen mit einem Buchstaben beginnen und können im Namen auch Ziffern sowie das Zeichen '-' enthalten. SQL-Schlüsselwörter dürfen nicht als Tabellennamen verwendet werden.⁴ Jede Tabelle wird beim Anlegen einem Schema zugeordnet. Wird kein Schema angegeben, wird die Tabelle im Schema des Benutzers, der die Tabelle erzeugt, angelegt.⁵

Innerhalb der Tabellendefinition werden die Spalten über ein oder mehrere Spaltendefinitionen festgelegt:

```
/—————\  
Spaltenname {Datentyp | Domänenname}  
    [NULL | NOT NULL]  
    [ [ CONSTRAINT Regelname ]  
        UNIQUE | PRIMARY KEY |  
        REFERENCES Tabellenname [(Spalten)] |  
        ON {DELETE [UPDATE] | UPDATE [DELETE]}  
            {CASCADE | SET NULL | SET DEFAULT | NO ACTION } |  
        CHECK (Bedingung) ...  
    ]  
    [ DEFAULT Wert ]  
—————/
```

Jede Spalte hat einen Namen sowie einen Datentyp, der einem Standarddatentyp oder einer selbstdefinierten Domäne entspricht. Zusätzlich können zu jeder Spalte eine Reihe weiterer Eigenschaften angegeben werden:

³Natürlich ist besser das Geburtsdatum in der Datenbank einzutragen, dies muss nicht jedes Jahr geändert werden.

⁴Diese Eigenschaften gelten für alle Objektnamen selbstdefinierter Objekte. Bei den folgenden Objektarten werden diese Namensbedingungen nicht mehr aufgeführt.

⁵Datenbankobjekte werden meist im Schema des angemeldeten Benutzers angelegt. Deshalb wird aus Gründen der Übersichtlichkeit in den nachfolgenden Syntaxbeschreibungen bei allen Datenbankobjekten der optionale Präfix 'Schema.' weggelassen. Immer wenn die Syntax einen Objektnamen wie Tabellen- oder Viewnamen zulässt, kann das Schema mit angegeben werden.

NOT NULL: Mit dieser Option werden Spalten definiert, für die ein Dateneintrag beim Einspeichern von Datensätzen obligatorisch sein soll. Jede Zeile der Tabelle muss damit in den so definierten Spalten einen Wert aufweisen.

NULL: Bei diesen Spalten sind NULL-Werte zulässig. Dies entspricht dem Default und wird deshalb meist nicht explizit angegeben.

PRIMARY KEY: Die Spalte entspricht dem Primärschlüssel. Damit ist meist automatisch die NOT NULL-Bedingung verknüpft.⁶ Dieser Zusatz kann nur bei einer Spalte angegeben werden, zusammengesetzte Primärschlüssel müssen im Bereich der Tabellenabhängigkeiten angegeben werden.⁷

UNIQUE: Die Werte der Spalte sind eindeutig, keine zwei Zeilen der Tabelle weisen bei diesem Attribut den selben Wert auf. Das Attribut ist damit ein Schlüsselkandidat.

Die Eindeutigkeit der Werte wird im SQL-Standard nur für Werte ungleich NULL verlangt, für die Behandlung von NULL-Werten gibt es keine Vorgaben. Entsprechend finden sich unterschiedliche Umsetzungen in den verschiedenen Datenbankprodukten. Der MS-SQL-Server als auch Informix verlangen, dass der Wert NULL auch nur einmal auftauchen darf. Oracle, PostgreSQL, MySQL oder SQLite erlauben beliebig viele NULL-Werte.

CHECK: Diese Option ermöglicht den Wertebereich der definierten Spalte einzuschränken, der in Form einer Bedingung hier hinterlegt sein kann.

REFERENCES: Bei Fremdschlüsselspalten wird hier die Tabelle und optional die Spalte angegeben, auf den die Fremdschlüsselspalte verweist. Wird keine Spalte angegeben, wird der Primärschlüssel der angegebenen Tabelle referenziert. Der Datentyp muss bei Fremdschlüsseln zur referenzierten Tabelle passen. Zusammengesetzte Fremdschlüssel müssen im Bereich der Tabellenabhängigkeiten angegeben werden. Gibt man selbst die referenzierten Spalten an, so ist neben dem Bezug zu einem Primärschlüssel auch ein Fremdschlüssel auf einen Schlüsselkandidaten (UNIQUE-Attribute) möglich.

Änderungen am referenzierten Wert oder sogar das Löschen des so referenzierten Datensatzes ist bei der Existenz eines referenzierenden Datensatzes nicht möglich. Durch die `ON DELETE` | `ON UPDATE`-Option kann aber eine derartige Änderung zugelassen werden. Das Ergebnis in der Datenbank ist dann abhängig von der angegebenen Aktion:

NO ACTION: Hierdurch wird ein Löschen bzw. Ändern verhindert, wenn abhängige Datensätze vorhanden sind. Dies ist der Default.

CASCADE: Änderungen am referenzierten Wert werden auf den referenzierenden Wert übertragen. Das Löschen des referenzierten Satzes hat das Löschen des referenzierenden zur Folge.

SET NULL: Änderungen des referenzierten Werts bewirken einen Eintrag von NULL bei den Fremdschlüsselattributen des referenzierenden Satzes.

SET DEFAULT: Änderungen des referenzierten Werts bewirken einen Eintrag der Defaultwerte bei den Fremdschlüsselattributen des referenzierenden Satzes.

CONSTRAINT: Bis auf NULL kann für jede der zuvor beschriebenen Bedingung durch diese Option ein Name angegeben werden. Dies erleichtert die Fehlersuche, wenn ein Wert aufgrund einer verletzten Bedingung nicht akzeptiert wird. In diesem Fall gibt das Datenbanksystem meist den Namen der Bedingung mit an, so dass man dann den Sachverhalt schnell überprüfen kann. Gibt man keinen Namen für die Bedingung an, vergibt das System einen internen Namen.

⁶Das DBVS DB2 von IBM verlangt explizit die Angabe von NOT NULL bei Primärschlüsselattributen. Auch die Standard-Norm verlangt diesen Zusatz, die Hersteller der meisten Produkte verzichten jedoch darauf.

⁷Während das 'reine' relationale Datenmodell die Angabe eines Primärschlüssels verlangt, erlauben SQL sowie die relationalen Datenbankprodukte auch Relationen ohne Primärschlüssel, die dann auch doppelte Datensätze enthalten dürfen. Relationen sind in ihrer Umsetzung in den Datenbankmanagementsystemen damit Multimengen, die auch Tupel mehrfach enthalten können.

DEFAULT: Der hier hinterlegte WERT wird als Defaultwert verwendet. Wird beim Einfügen eines Datensatzes kein Wert angegeben, wird der Defaultwert als Spaltenwert verwendet. Der Standard-Defaultwert ist NULL. Auch eine Systemvariablen, wie beispielsweise USER bei Oracle, ist ein möglicher Defaultwert, der Wert entspricht dem Loginnamen des angemeldeten Benutzers.

Beispiele:

```
⇒ CREATE TABLE Abteilung (
    ANR           CHAR(4)          PRIMARY KEY,
    Name          VARCHAR(32)      NOT NULL,
    Budget        NUMERIC(13,2)
)
⇒ CREATE TABLE Pers (
    PNR           CHAR(10)         PRIMARY KEY,
    Name          VARCHAR(32)      NOT NULL,
    Abteilung     CHAR(4)          REFERENCES Abteilung(ANR),
    Vorgesetzter   CHAR(10)         REFERENCES Pers(PNR),
    Gehalt        DGehalt
)
```

Die bisherigen Integritätsbedingungen beziehen sich immer auf die Spalte, bei der die jeweilige Bedingung angegeben ist. Integritätsbedingungen, die mehrere Spalten betreffen, sind als eigenständige Elemente der Tabellendefinition als Tabellenbedingungen zu definieren:

```
/—————\ [CONSTRAINT Regelname]
{ UNIQUE | PRIMARY KEY }(Spaltenname [, Spaltenname]...) |
FOREIGN KEY (Spaltenname [, Spaltenname]...)
    REFERENCES Tabelle [(Spaltenname [, Spaltenname]...)]
    [ON {DELETE [UPDATE] | UPDATE [DELETE] }
        {CASCADE | SET NULL | SET DEFAULT | NO ACTION } ] |
CHECK (Bedingung)
—————/
```

PRIMARY KEY: Besteht der Primärschlüssel aus mehreren Attributen, kann dieser hier angegeben werden, indem die Attribute in einer Liste angegeben werden. Aber auch Primärschlüssel aus einem Attribut können so definiert werden.

UNIQUE: Analog zum Primärschlüssel können andere Schlüsselkandidaten, die mehrere Attribute umfassen, in einer Liste angegeben werden.

CHECK: Diese Option ermöglicht Bedingungen an die Werte zu formulieren, die hier auch mehrere Attribute umfassen. So kann beispielsweise mit `check (Budget > Gehaltssumme)` ein Plausibilitätstest der Beiträge einer entsprechenden Tabelle erfolgen.

FOREIGN KEY: Zusammengesetzte Fremdschlüssele sind analog aufgebaut. Bei der referenzierenden Tabelle sind dadurch ebenfalls mehrere Spaltennamen zulässig. Die ON DELETE bzw. ON UPDATE Möglichkeiten sowie Fremdschlüssele auf UNIQUE-Attribute können auch hier angegeben werden.

CONSTRAINT: Für jede der zuvor beschriebenen Bedingung kann durch diese Option ein Name angegeben werden. Gibt man keinen Namen für die Bedingung an, vergibt das Datenbanksystem einen internen Namen.

Beispiele:

```

⇒ CREATE TABLE Flughafen (
    FHKuerzel      CHAR(3)          PRIMARY KEY,
    Name           VARCHAR(32),
    Ort            VARCHAR(32)        NOT NULL,
    Landebahnen    INT              NOT NULL
)
⇒ CREATE TABLE Fluggesellschaft (
    FGKuerzel     CHAR(2),
    Name          VARCHAR(32)        NOT NULL
                                UNIQUE,
    BasisFH       CHAR(3),
    PRIMARY KEY (FGKuerzel),
    FOREIGN KEY (BasisFH) REFERENCES Flughafen(FHKuerzel)
)
/* Beide Schlüsselinformationen hätten auch direkt beim
   Attribut hinterlegt werden können.*/
⇒ CREATE TABLE Flugzeug (
    FKZ           CHAR(8)          PRIMARY KEY,
    Hersteller    VARCHAR(32),
    Typ           VARCHAR(8),
    FGKuerzel    CHAR(2),
    Sitzplaetze  INT,
    FOREIGN KEY (FGKuerzel)
                                REFERENCES Fluggesellschaft (FGKuerzel)
)
⇒ CREATE TABLE Pilot (
    PNR           CHAR(10)         PRIMARY KEY,
    Name          VARCHAR(32)        NOT NULL,
    WOrt          VARCHAR(32),
    GebDat       DATE,
    UTermin      DATE             NOT NULL,
    CHECK (UTermin >= SYSDATE AND UTermin <= SYSDATE + 90)
)
/* SYSDATE enthält bei ORACLE das aktuelle Datum */
⇒ CREATE TABLE Flug (
    PNR           CHAR(10)         NOT NULL,
    FKZ           CHAR(8),
    Flugdatum    TIMESTAMP,
    Start         VARCHAR(32),
    Ziel          VARCHAR(32),
    Passagiere   INT,
    PRIMARY KEY (FKZ, Flugdatum),
    FOREIGN KEY (FKZ) REFERENCES Flugzeug(FKZ),
    FOREIGN KEY (PNR) REFERENCES Pilot(PNR)
)

```

Bei einer gut gestalteten Datenbank sind die Tabellen und die Spalten auch gut dokumentiert, so dass die Benutzer der Tabellen die Bedeutung der Elemente bei Bedarf nachlesen können. Diese Dokumentation kann mit dem COMMENT-Befehl direkt in der Datenbank erfolgen, die Kommentare werden in Tabellen des Systemkatalogs abgelegt und können daraus ausgelesen werden (siehe hierzu auch Abschnitt 5.8).

/_____\
COMMENT ON

```
{TABLE {Tabellenname | Viewname} |
    COLUMN {Tabellenname.Spalte | Viewname.Spalte}}
    IS 'Kommentar'
\-----/
```

Beispiele:

```
⇒ COMMENT ON TABLE Flug
    IS 'Tabelle zum Abspeichern aller Fluginformationen'
⇒ COMMENT ON COLUMN Flug.Flugdatum
    IS 'Datum und Uhrzeit des Starts'
```

5.2.5 Indizes

Der Einsatz von Indizes ist für die Arbeit mit relationalen Datenbanken nicht unbedingt erforderlich, sie sind aber zur Steigerung der Ausführungsgeschwindigkeit sehr nützlich. Indizes erlauben dabei ein schnelleres Auffinden eines Datensatzes bzgl. eines bestimmten Attributwerts oder Wertebereichs. Weitere Informationen zu Indizes finden sich in Kapitel 8.

5.2.6 Synonyme

Ein Synonym ist ein Alias für eine Tabelle. Es dient primär zur Verkürzung der Eingabe eines Tabellennamens, insbesondere wenn bei der Tabelle der Schemaname mit angegeben werden muss.

```
/-----\
CREATE [PUBLIC] SYNONYM Synonym FOR [Schema.]Object
\-----/
```

Synonyme, die PUBLIC sind, stehen allen Benutzern zur Verfügung.

Beispiel:

```
⇒ CREATE SYNONYM PERS FOR BUCHHALTUNG.PERSONAL
    In allen SQL Anfragen, in denen die PERSONAL-Tabelle im Schema BUCHHALTUNG eingesetzt wird, kann nun PERS ohne Angabe des Schemas verwendet werden.
```

5.2.7 Änderungen an Tabellendefinitionen

Eine einmal definierte Tabellenstruktur ist auch Änderungen unterworfen. Wenn auch beim Datenbankentwurf soviel Sorgfalt in das Design investiert werden sollte, dass die einmal erstellten Tabellen unverändert bleiben können, wird dies nicht auf Dauer zu erwarten sein. So wachsen häufig die Anforderungen oder neue Unternehmensdaten fallen an, an die man zu Beginn nicht denken konnte. Der Befehl zum Ändern einer Tabellendefinition ist ALTER TABLE.

```
/-----\
ALTER TABLE Tabellenname
{ ADD [COLUMN] Spaltendefinition
| MODIFY [COLUMN] Sp {DEFAULT Wert | DROP DEFAULT}
| DROP [COLUMN] Sp [RESTRICT | CASCADE]
| ADD Tabellenconstraint
| DROP CONSTRAINT constraint }
\-----/
```

Beispiele:

⇒ `ALTER TABLE Pers ADD Vertreter INT UNIQUE`

Die Spalte Vertreter wird in die Tabelle Pers aufgenommen. Die vorhandenen Datensätze in der Tabelle Pers bekommen den Wert NULL für das neue Attribut. NOT NULL-Spalten können damit nur eingefügt werden, wenn die Tabelle leer ist.

⇒ `ALTER TABLE Pers ADD`

`FOREIGN KEY (Vertreter) REFERENCES Pers (PNR)`
Die neue Spalte ist ein Fremdschlüssel.

⇒ `ALTER TABLE Pers MODIFY COLUMN Vertreter DEFAULT 'a18'`

Die neue Spalte bekommt für neue Datensätze einen Defaultwert.

⇒ `ALTER TABLE Pers DROP COLUMN Vertreter RESTRICT`

Die neue Spalte wird gelöscht. RESTRICT führt zur Zurückweisung der Operation, wenn das Attribut in einer View (siehe hierzu Abschnitt 5.9.1) oder einer Integritätsbedingung referenziert wird. CASCADE dagegen erzwingt die Folgelöschung aller Views und CHECK-Klauseln, die von dem Attribut abhängen. Das letzte Attribut einer Relation kann nicht gelöscht werden.

5.2.8 Löschen von Objekten

Alle Objekte, die mit einem CREATE-Befehl erzeugt werden, können mit einem passenden DROP-Befehl aus der Datenbank entfernt werden. Bei Verwendung der CASCADE-Option lassen sich abhängige Objekte (beispielsweise Views und Referenzconstraints) mitentfernen. RESTRICT verhindert das Löschen, wenn das Objekt noch referenziert wird.

```
/—————\  
DROP  
{ TABLE Tabellename  
| VIEW Viewname  
| DOMAIN Domainname  
| INDEX Indexname  
| SCHEMA Schemaname}  
[RESTRICT | CASCADE]  
\—————/
```

Beispiele:

⇒ `DROP DOMAIN DALter`

⇒ `DROP TABLE Pers RESTRICT`

5.3 Datenbankanfragen

Der Hauptzweck eines Datenbanksystems besteht darin, Informationen zu speichern, die jederzeit auf einfache Weise wieder verfügbar gemacht werden können. Dies soll zum einen für die Verarbeitung innerhalb einer Anwendung und andererseits für einen direkten Zugriff durch autorisierte Personen möglich sein. Der SQL-Befehl hierzu ist der SELECT-Befehl mit dem Daten abgefragt, sortiert, gruppiert und verrechnet werden können. Die Ausgabe erfolgt in Form einer Tabelle, der Ergebnistabelle.

```
/—————\  
SELECT [DISTINCT | ALL]  
{ * |
```

```

{ Ausdruck [[AS] ColAlias] |
{Tabname | Viewname | TabAlias}.* |
{Tabname | Viewname | TabAlias}.Spalte | [[AS] ColAlias]}
[, Ausdruck [[AS] ColAlias] |
{Tabname | Viewname | TabAlias}.* |
{Tabname | Viewname | TabAlias}.Spalte | [[AS] ColAlias]]...
FROM {Tabname | Viewname} [TabAlias]
[, {Tabname | Viewname} [TabAlias]]...
[ WHERE Bedingung ]
[ GROUP BY Ausdruck [, Ausdruck] ... [HAVING Bedingung] ]
[ ORDER BY {Spalte | Position} [ASC | DESC]
[, {Spalte | Position} [ASC | DESC]] ...]
[FOR UPDATE [OF UpdateSpalten]]
\—————/

```

5.3.1 Die Grundbausteine SELECT und FROM

Die einfachste Form von Datenbankanfragen benötigt nur den SELECT- und den FROM-Teil. Nach dem Schlüsselwort SELECT werden dabei alle Spaltennamen angegeben, die man in die Ergebnistabelle aufnehmen möchte. Sind alle Spalten der in der FROM-Klausel angegebenen Relationen gewünscht, kann man dies mit * abkürzen. Weiterhin bietet dieser Teil die Optionen ALL und DISTINCT. Letzteres bewirkt, dass doppelte Zeilen in der Ergebnistabelle nicht mit aufgenommen werden.⁸ ALL lässt dies zu, wobei ALL der Default ist und deshalb meist nicht explizit aufgeführt wird.

Nach dem Schlüsselwort FROM wird die Relation aufgeführt aus der die Daten gelesen werden sollen. Damit lassen sich schon Übersichtsanfragen realisieren.

Beispiele:

- ⇒ Liste aller Flughäfen mit allen Informationen:
SELECT * FROM Flughafen
- ⇒ Liste aller Ziele von Flügen
SELECT DISTINCT Ziel FROM Flug
- ⇒ Liste aller Flugverbindungen
SELECT DISTINCT Start, Ziel FROM Flug

5.3.2 Auswahl von Zeilen

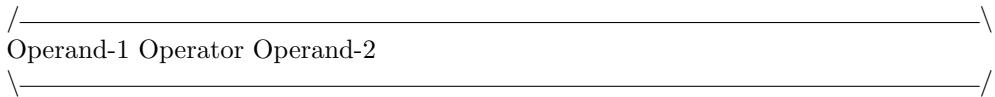
Soll das Ergebnis nur bestimmte Zeilen enthalten, muss im SELECT angegeben werden, welche Zeilen gewünscht sind. Hierzu kann eine Bedingung im WHERE-Teil der Anfrage angegeben werden. Alle Datensätze, die die Bedingung erfüllen, werden dann in das Ergebnis aufgenommen.

Um allen Anforderungen einer Datenbankrecherche zu genügen, kann eine Suchbedingung als einfacher Vergleich oder zusammengesetzte Bedingung formuliert und mit weiteren Abfragen verbunden sein. Dazu verfügt SQL über eine Reihe von Operatoren, die nachfolgend vorgestellt werden.

⁸Datenbanksysteme ermitteln teilweise die doppelten Werte, indem alle Werte sortiert werden, danach stehen die doppelten hintereinander. Daher bewirkt ein DISTINCT teilweise automatisch eine Sortierung. Darauf sollte man sich aber nicht verlassen. So kann Oracle seit der Version 10.2 für das Löschen doppelter Werte auch alternative Algorithmen einsetzen, wodurch nach einem Update die Sortierung nicht automatisch mehr gegeben war.

5.3.2.1 Einfache Vergleiche

Ein einfacher Vergleich hat folgenden Aufbau:



Operanden können dabei sein:

- ▶ Spaltennamen von Tabellen oder Views, die in der FROM-Klausel genannt werden
- ▶ Konstanten (Literale)
- ▶ Arithmetische Ausdrücke (Operatoren +,-,*,/ mit Klammerung sowie diverse Funktionen, auch für Zeichenketten und Datumsberechnungen)
- ▶ Systemvariablen (Beispiel: SYSDATE bei Oracle mit aktuellem Datum)
- ▶ Unterabfragen

Vergleichsoperatoren sind:

=:	gleich
>:	größer
<:	kleiner
>=:	größer oder gleich
<=:	kleiner oder gleich
!=,<>:	ungleich

Beispiele:

- ⇒ Welche Piloten (PNR) fliegen nach Rom?

```
SELECT PNR
  FROM Flug
 WHERE Ziel = 'Rom'
```

- ⇒ Welche Mitarbeiter (PNR, Name) verdienen mit einer 5-%igen Gehaltserhöhung weniger als 40000 Euro?

```
SELECT PNR, Name
  FROM Pers
 WHERE Gehalt * 1.05 < 40000
```

Innerhalb einer WHERE-Bedingung können auch mehrere Suchkriterien angegeben werden, die mit den Junktoren AND, OR und NOT sowie unter Zuhilfenahme von Klammerungen aufgebaut sein können. NOT hat dabei die höchste Priorität, gefolgt von AND.

Beispiel:

- ⇒ Welche Piloten (PNR) flogen am 12.12.2000 von Frankfurt nach Rom oder Neapel?

```
SELECT PNR
  FROM Flug
 WHERE Flugdatum = '12.12.2000'
       AND Start = 'Frankfurt'
       AND (Ziel = 'Rom' OR Ziel = 'Neapel')
```

5.3.2.2 Der Bereichs-Operator BETWEEN

Für Vergleiche mit einem Wertebereich stellt SQL den Operator BETWEEN zur Verfügung.



Der Bereich ist einschließlich der Grenzen definiert.

`a BETWEEN b AND c` ist damit gleichbedeutend zu `a >= b AND a <= c`.

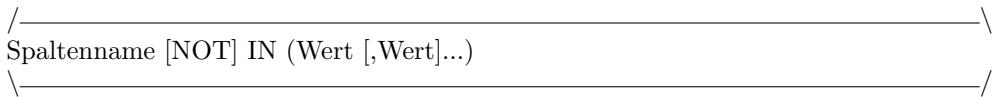
Beispiel:

⇒ Welche Mitarbeiter verdienen nicht zwischen 50000 und 60000 Euro?

```
SELECT *  
FROM Pers  
WHERE Gehalt NOT BETWEEN 50000 AND 60000
```

5.3.2.3 Der Operator IN

Wenn bei einer Suchbedingung eine Folge von Werten für ein Attribut OR-verknüpft zu testen ist, erlaubt der IN-Operator eine kompaktere Darstellung der Anfrage. Man gibt dabei beim Vergleich nach dem IN-Operator eine Liste der Alternativen an.



Beispiel:

⇒ Welche Piloten (PNR) flogen nach Rom, Mailand, Turin oder Neapel?

```
SELECT PNR  
FROM Flug  
WHERE Ziel IN ('Rom', 'Mailand', 'Turin', 'Neapel')
```

5.3.2.4 Zeichenkettenähnlichkeiten

Die Werte einer Zeichenkettenattribute können mit dem LIKE-Operator gemäß einem Muster verglichen werden. Das Muster wird in Form einer Zeichenkette mit Platzhaltern angegeben. Mögliche Platzhalter sind dabei '%', das für eine beliebige (auch leere) Folge von Zeichen steht und '_', das für ein beliebiges Zeichen steht.



Wenn ein Platzhalterzeichen im Suchbegriff vorkommt und damit dieses Zeichen in der Maske verwendet werden soll, muss ein ESCAPE-Zeichen angegeben werden. Ein ESCAPE-Zeichen vor einem Platzhalter gibt an, dass das Platzhaltersymbol nicht als Platzhaltersymbol verwendet werden soll.

LIKE-Klausel	Wird erfüllt von
Name LIKE '%Schmi%'	'H.-W. Schmitt' 'Schmitt, H.-W.' 'BauSchmied', 'Schmitz'
ANR LIKE '_7%'	Abteilungen, die an zweiter Stelle eine 7 haben (K70, J7, M788888)
Name NOT LIKE '%-%'	Namen ohne Bindestrich
Name LIKE '%*_%' ESCAPE '*'	Alle Namen, die ein '_' beinhalten

Tabelle 5.2: Der LIKE-Operator

5.3.3 Sortieren

Eine Sortierung der Ergebnistabelle einer Anfrage nach einer oder mehreren Spalten wird durch die Klausel ORDER BY ermöglicht.

```
/—————\  
ORDER BY {Spalte | Spaltennummer} [ASC | DESC]  
[ , {Spalte | Spaltennummer} [ASC | DESC]] ...  
—————/
```

Die Spalten, nach denen sortiert werden soll, müssen in der Spaltenliste der Selektion nach dem Schlüsselwort SELECT enthalten sein. Die Sortierung kann nach mehreren dieser Spalten erfolgen, wobei die Hierarchie nach rechts abnimmt. Es wird zuerst nach der Spalte mit der höchsten Rangfolge sortiert. Treten hier gleiche Werte auf, wird innerhalb dieser Werte nach der nächsten Spalte sortiert und so fort. Die Spalten können numerisch, alphanumerisch oder Datumswerte beinhalten.

Wenn in der SELECT-Anweisung keine Spaltenliste aufgeführt ist, weil beispielsweise Berechnungen durchgeführt werden, kann eine Referenznummer entsprechend der Spaltenanordnung in der Ergebnistabelle herangezogen werden. Diese Alternative ist natürlich auch für solche Spalten erlaubt, die benannt werden können. Weiterhin kann für jede Spalte angegeben werden, ob sie aufsteigend (ASC) oder absteigend (DESC) sortiert werden soll. Ohne Angabe wird aufsteigend sortiert. Null-Werte stehen in der Rangfolge oben.

Beispiele:

⇒ Welche Mitarbeiter (Name, Gehalt) arbeiten in Abteilung 'K7' sortiert nach Gehalt?

```
SELECT Name, Gehalt
FROM Pers
WHERE ANR = 'K7'
ORDER BY Gehalt
```

⇒ Welche Mitarbeiter (ANR, Name) verdienen mehr als 30000 Euro sortiert nach ANR
absteigend und Name aufsteigend?

```
SELECT ANR,Name
FROM Pers
WHERE Gehalt > 30000
ORDER BY ANR DESC, Name
```

5.3.4 Ausdrücke

In Abschnitt 5.3.2.1 wurde im Rahmen der Syntaxbeschreibung der Begriff 'Ausdruck' verwendet, ohne dass dieser präziser erläutert wurde. Ein Ausdruck ergibt bei der Auswertung immer einen Wert. In den bisherigen Beispielen wurden als Ausdrücke beispielsweise Konstanten verwendet. So

hat die Konstante 'Gregory' bei der Auswertung den Zeichenkettenwert 'Gregory'. Auch Spaltennamen wurden in den vorherigen Beispielen schon für Ausdrücke verwendet. Die Auswertung eines Spaltenamens ergibt dann den Wert dieser Spalte des aktuellen Datensatzes.

SQL erlaubt diverse weitere Möglichkeiten bei Wertangaben in Form von Ausdrücken. So stehen für numerische Werte direkt die Grundrechenoperationen +, -, *, / zur Verfügung. Zeichenketten können mit || konkatiniert werden. Zudem stellen die Datenbankverwaltungssysteme diverse Grundfunktionen zur Verfügung. Dazu gehören beispielsweise die trigonometrischen Funktionen wie `sind`, `cos`, `tan` oder Datumsberechnungsfunktionen sowie Zeichenkettenfunktionen wie beispielsweise `upper` zur Umwandlung einer Zeichenkette in Großbuchstaben zur Verfügung. Der Funktionsumfang und teilweise auch die Namen der Funktionen sind datenbankspezifisch, so dass man hier bei dem jeweiligen Datenbankprodukt nachschauen muss, welche Funktionen angeboten werden und wie diese heißen.

Ausdrücke können überall verwendet werden, wo Werte erwartet werden. Mögliche Verwendungsbereiche sind damit:

- ▶ Spaltenselektion nach SELECT
- ▶ Vergleichswert in WHERE-Klauseln
- ▶ ORDER-BY-Klauseln
- ▶ Werte für neue Datensätze (INSERT)
- ▶ Neue Spaltenwerte bei Änderungen (UPDATE)

5.3.4.1 Selektieren und Benennung von Ausdrücken

Bei der Selektion können direkt auch Ausdrücke selektiert werden, so dass eine Anfrage direkt auch Berechnungsergebnisse liefern kann. Das Berechnungsergebnis wird dabei für jeden Datensatz ermittelt. Diese Berechnungsergebnisspalten der Anfrage haben keinen vordefinierten Namen, es sei denn, man vergibt ihn explizit innerhalb des SELECT-Teils durch die Option [[AS] ColAlias]

Beispiel:

```
⇒ SELECT NAME,
      'Wunschgehalt: ' AS Text,
      Gehalt * 1.2 AS Jahresgehalt
      Gehalt * 1.2 / 12 AS Monatsgehalt
   FROM Pilot
```

5.3.4.2 CASE-Ausdrücke

Mit CASE-Ausdrücken können Fallunterscheidungen in Anfragen aufgenommen werden. So können hierüber beispielsweise kodierte Werte in lesbare Form überführt werden oder eine Funktionsausführung kann abhängig von Bedingungen integriert werden.

```
/—————\
CASE [selector]
    WHEN ausdruck THEN result
    [WHEN ausdruck THEN result] ...
    [ELSE result]
END
\—————/
```

Die Syntax unterstützt dabei zwei Varianten. Bei einfachen CASE-Ausdrücken wird als Selektor ein Wert angegeben, der mit den in den WHEN-Klauseln angegebenen Wertausdrücken verglichen wird. Bei Übereinstimmung wird der dazugehörige WHEN-Teil ausgeführt. Ist keine Bedingung erfüllt, wird der Wert als Ergebnis verwendet, der in einer optionalen ELSE-Klausel steht. Hat das CASE keine ELSE-Klausel und ist keine Bedingung erfüllt, liefert der CASE-Ausdruck den Wert NULL.

Beispiel:

```
⇒ SELECT MNR, CASE note
    WHEN 1 THEN 'sehr gut'
    WHEN 2 THEN 'gut'
    WHEN 3 THEN 'befriedigend'
    WHEN 4 THEN 'ausreichend'
    ELSE 'nicht ausreichend'
END
FROM Student
```

Bei den einfachen Case-Ausdrücken wird nur auf Gleichheit mit den in den WHEN-Klauseln angegebenen Werten geprüft. Für komplexere Vergleiche kann der sogenannte durchsuchte CASE-Ausdruck verwendet werden. Dieser arbeitet ohne Selektor, in den WHEN-Klauseln kann dafür eine vollständige Bedingung stehen. Die WHEN-Klausel, deren Bedingungsauswertung als erstes erfüllt ist, definiert dann den Rückgabewert.

Beispiel:

```
⇒ SELECT MNR, CASE
    WHEN note <= 1.5 THEN 'sehr gut'
    WHEN note <= 2.5 THEN 'gut'
    WHEN note <= 3.5 THEN 'befriedigend'
    WHEN note <= 4 THEN 'ausreichend'
    ELSE 'nicht ausreichend'
END
FROM Student
```

5.3.4.3 Datentyp-Umwandlung (casting)

Die Umwandlung eines Werts von einem Datentyp in einen anderen kann implizit oder explizit erfolgen. Vergleicht man beispielsweise der Ausdruck `5 - '1'` verwendet, wird die Zeichenkette `'1'` implizit in die Zahl 1 umgewandelt, so dass die Berechnung durchgeführt werden kann. Schwieriger wird es bei Vergleichen der Form `Gehalt = '1000'`. Auch hier findet eine implizite Typumwandlung statt, die jedoch nicht eindeutig ist. Hier könnte der Zeichenkettenwert `'1000'` in eine Zahl umgewandelt werden oder der numerische Gehaltswert könnte vor dem Vergleich in eine Zeichenkette umgewandelt werden.

Sicherer ist es hier, die Typumwandlung selbst explizit über Casting-Funktionen zu steuern. Die Casting-Funktionen sind dabei im SQL-ISO-Standard definiert, die Datenbankhersteller setzen hier diesen Standard jedoch nicht durchgängig und einheitlich um, so dass für die Typumwandlungsfunktionen die Dokumentation des jeweiligen Datenbankherstellers herangezogen werden muss.

5.3.5 Verknüpfung von Tabellen, Join

Die Informationen, die in den bisherigen Abfragen gewonnen wurden, stammen jeweils nur aus einer Tabelle. Für Anfragen, die Informationen aus mehreren Tabellen zusammenführen, kann man in der FROM-Klausel auch mehrere Tabellen angeben. Dabei kann es jedoch vorkommen,

dass ein Attributname in mehreren Tabellen vorkommt. In diesem Fall ist vor dem Attributnamen der Tabellenname anzugeben, so dass eine eindeutige Zuordnung erfolgen kann.

Beispiel:

```
⇒ SELECT Flughafen.Name, Pilot.Name
   FROM Flughafen, Pilot WHERE .....
```

Mit dem Einsatz von Aliasnamen für die Tabellen kann dies verkürzt werden. Hierbei wird in der FROM-Klausel hinter einem Tabellennamen der Aliasname angegeben, der dann anstatt des Tabellennamens verwendet werden kann.

Beispiel:

```
⇒ SELECT F.Name, P.Name
   FROM Flughafen F, Pilot P WHERE .....
```

5.3.5.1 Verknüpfen von Tabellen mittels der WHERE-Klausel

Bei der Verknüpfung von zwei oder mehr Tabellen, genannt Join, ist eine Besonderheit zu beachten, die an folgenden Testtabellen demonstriert werden soll.

Tabelle A		Tabelle B	
A1	A2	B1	B2
10	150	20	210
20	170	30	274

Wenn diese zwei Tabellen ohne weitere Einschränkung zusammengeführt werden, entsteht eine Ergebnistabelle, die dem kartesischen Produkt entspricht. Jede Zeile der einen Tabelle wird mit jeder Zeile der anderen Tabelle verknüpft.

A1	A2	B1	B2
10	150	20	210
10	150	30	274
20	170	20	210
20	170	30	274

Um zu einem sinnvollen Ergebnis zu kommen, bedarf es eines Verknüpfungskriteriums, das eine Beziehung zwischen diesen beiden Tabellen herstellt. Bei diesen Kriterium handelt es sich um Spalten in den Tabellen, die vom gleichen Datentyp sind und vergleichbare Daten beinhalten. Dies sind meist die Spalten, die Fremd-/Primärschlüsselbeziehungen entsprechen. Die Beziehung wird durch die WHERE-Klausel in Form einer Join-Bedingung hergestellt, die angibt wie die Spalten in Beziehung gestellt werden. Ein `SELECT * FROM A,B WHERE A.A1 = B.B1` ergibt dadurch beispielsweise folgende Tabelle:

A1	A2	B1	B2
20	170	20	210

Da die Join-Bedingung in diesem Fall durch den Gleichheitsoperator ausgedrückt ist, spricht man hier auch von einem Equi-Join.

Beispiele:

⇒ Finden Sie alle Namen von Fluggesellschaften, die Flugzeuge von Boeing besitzen.

```
SELECT FG.Name
FROM Flugzeug FZ, Fluggesellschaft FG
WHERE FZ.Hersteller = 'Boeing' AND
      FZ.FGKuerzel = FG.FGKuerzel
```

⇒ Finden Sie alle Piloten (PNR), die an einem Ort wohnen, an dem sie gestartet sind.

```
SELECT P.PNR
FROM Pilot P, Flug F
WHERE P.PNR = F.PNR AND
      P.WOrt = F.Start
```

⇒ Finden Sie alle Piloten (Name, WOrt), die bei der Lufthansa einen A300 geflogen sind.

```
SELECT P.Name, P.WOrt
FROM Pilot P, Flug F, Flugzeug FZ, Fluggesellschaft FG
WHERE P.PNR = F.PNR AND
      F.FKZ = FZ.FKZ AND
      FZ.FGKuerzel = FG.FGKuerzel AND
      FG.Name = 'Lufthansa' AND
      FZ.Typ = 'A300'
```

⇒ Finden Sie die Angestellten, die mehr als ihre direkten Vorgesetzten verdienen (Alles vom Angestellten + Name des Vorgesetzten).

```
SELECT A.* , M.Name
FROM Pers A, Pers M
WHERE A.Vorgesetzter = M.PNR AND
      A.Gehalt > M.Gehalt
```

Das *-Symbol innerhalb der SELECT-Attributliste wird auf eine Tabelle eingeschränkt, indem man den Tabellennamen mit einem Punkt vor das *-Symbol stellt.

⇒ Welche Piloten (PNR) wohnen an einem Flughafen?

```
SELECT P.PNR
FROM Pilot P, Flughafen F
WHERE P.WOrt = F.FOrt
```

5.3.5.2 Alternative Join-Formulierungen in SQL2

In der FROM-Klausel wird angegeben, auf welche Tabellen zugegriffen werden soll. Für die Lesbarkeit von Anweisungen kann es dabei nützlich sein, wenn innerhalb der FROM-Klausel auch direkt angegeben werden kann, wie die Tabellen zu verknüpfen sind. Diese Möglichkeit hat man mit SQL2 in die Syntax aufgenommen, wobei mehrere Alternativen aufgenommen wurden.

Bisher:

```
SELECT * FROM Pilot P, Flug F WHERE P.PNR = F.PNR
```

Verbund über alle Attribute, die in beiden Tabellen vorkommen (Namensgleichheit):

```
SELECT * FROM Pilot NATURAL JOIN Flug
```

Equijoin mit Attributangabe, Attribute müssen in beiden Tabellen vorkommen:

```
SELECT * FROM Pilot JOIN Flug USING (PNR)
```

Join mit Vergleichsangabe:

```
SELECT * FROM Pilot P INNER JOIN Flug F ON P.PNR=F.PNR9
```

⁹Bei einigen Systemen ist der Zusatz INNER nicht unbedingt erforderlich, im SQL-Standard ist die Angabe nicht obligatorisch. Aus Gründen einer direkt erkennbaren Unterscheidung zu den folgenden offenen Verbunden wird die Verwendung jedoch empfohlen.

5.3.5.3 Offene Verbunde

Bei einem Verbund zwischen zwei Tabellen werden die Datensätze miteinander verbunden, die die Verbund-Bedingung erfüllen. Erfüllt ein Datensatz einer Relation mit keinem anderen Datensatz der Relation, mit der der Verbund durchgeführt wird, so taucht dieser Datensatz im Join-Ergebnis nicht auf. Dies ist sehr gut am anfänglichen Beispiel von Abschnitt 5.3.5.1 gut erkennbar, nur jeweils ein Datensatz der beiden Tabellen finden sich im dargestellten Join-Ergebnis.

Möchte man alle Datensätze der Ausgangstabellen in Anfrageergebnis erhalten, kann man über offene Verbunde erreichen, dass für Datensätze, die über die Bedingung kein Join-Partner existiert, ein künstlicher Join-Partner aus NULL-Werten erzeugt wird, mit dem dann der Datensatz verknüpft wird und so im Ergebnis der Anfrage vorkommt. In der SQL2-Join-Syntax werden hierfür anstatt der Angabe INNER JOIN die Schlüsselwörter FULL OUTER JOIN verwendet, wobei das Schlüsselwort OUTER hier optional ist. Möchte man, dass nur von einer der beiden Tabellen Datensätze ohne Join-Partner die NULL-Join-Partner erzeugt werden, kann dies über die Angabe von LEFT OUTER JOIN bzw. RIGHT OUTER JOIN erreicht werden. Datensätze ohne Join-Partner der anderen Seite sind dann im Anfrageergebnis nicht zu finden. Im Beispiel:

Tabelle A		Tabelle B	
A1	A2	B1	B2
10	150		
20	170		

SELECT * FROM A INNER JOIN B				SELECT * FROM A LEFT OUTER JOIN B				SELECT * FROM A RIGHT OUTER JOIN B			
A1	A2	B1	B2	A1	A2	B1	B2	A1	A2	B1	B2
20	170	20	210	20	170	20	210	20	170	20	210
				10	150	-	-	10	150	-	-
A1	A2	B1	B2	A1	A2	B1	B2	A1	A2	B1	B2
20	170	20	210	20	170	20	210	20	170	20	210
10	150	-	-	10	150	-	-	10	150	-	-
-	-	30	274	-	-	30	274	-	-	30	274

5.3.5.4 Cross Join

Wie zuvor dargestellt kommt bei einer Anfrage, bei der in der FROM-Klausel mehrere Tabellen mit Komma getrennt angegeben werden, das kartesische Produkt der angegebenen Tabellen heraus. Dies ist dem Befehl intuitiv nicht ansehbar, der Sachverhalt stammt aus der von Edgar Codd definierten Relationenalgebra als mathematische Basis zum Umgang mit und zur Darstellung von Anfragen auf Relationen. Mit SQL2 hat man den Cross-Joint eingeführt, der auch das kartesische Produkt ermittelt, nun ist es dem Befehl aber besser anzusehen:

Bisher:

```
SELECT * FROM A, B
```

Cross Join:

```
SELECT * FROM A CROSS JOIN B
```

5.3.6 Gruppenfunktionen

Neben den Funktionen für Standardberechnungen gibt es in SQL auch so genannte Gruppenfunktionen, die Berechnungen über eine Menge von Datensätzen erlauben. Die Funktionen sind:

AVG: Mittelwert

COUNT: Anzahl

MAX: Maximum

MIN: Minimum

SUM: Summe

```
/-----\
{AVG | SUM} ([DISTINCT] Spaltenname) | Math.Ausdruck) |
COUNT (* | [DISTINCT] Spaltenname) |
{MAX | MIN (Spaltenname | Math.Ausdruck)
\-----/
```

Da die Funktionen Spalten- bzw. Gruppenbearbeitungen vornehmen, dürfen sie in der WHERE-Klausel, die Bedingungen für einen Datensatz formuliert, nicht verwendet werden. AVG und SUM sind nur auf numerischen Ausdrücken definiert, MIN und MAX können für jeden Grunddatentyp eingesetzt werden.

COUNT(*) zählt die Gesamtanzahl an Zeilen, während ein COUNT(DISTINCT Spalte) nur die Anzahl an unterschiedlichen Werten ermittelt. Ein COUNT ohne DISTINCT ermittelt die Anzahl der Zeilen, die für dieses Attribut einen Wert haben, also nicht NULL sind.

Beispiele:

- ⇒ Bestimmen Sie das Durchschnittsgehalt der Angestellten.
`SELECT AVG(Gehalt) FROM Pers`
- ⇒ Bestimmen Sie das durchschnittliche Monatsgehalt der Angestellten.
`SELECT AVG(Gehalt/12) FROM Pers`
- ⇒ Wieviele Piloten flogen nach 'Rom'?
`SELECT COUNT(DISTINCT PNR) FROM Flug`
`WHERE ZIEL = 'Rom'`
- ⇒ Wie viele Angestellte haben einen Vorgesetzten?
`SELECT COUNT(Vorgesetzter) FROM Pers`
- ⇒ Wie viele Angestellte haben keinen Vorgesetzten?
`SELECT COUNT(PNR) - COUNT(Vorgesetzter) FROM Pers`
- ⇒ Wie viele Angestellte sind Vorgesetzte?
`SELECT COUNT(DISTINCT Vorgesetzter) FROM Pers`

5.3.7 Gruppenbildung

Ein häufig auftretendes Problem ist die Gruppenverarbeitung. Mit der GROUP BY-Klausel lässt sich in SQL recht einfach eine Gruppierung von Daten erreichen, die gleiche Werte in einer oder mehreren Spalten aufweisen. Mit der zusätzlich speziell für Gruppierungen gedachten HAVING-Klausel ist es möglich, Gruppenzeilen nach einer Bedingung auszuwählen.

```
/-----\
GROUP BY Ausdruck [, Ausdruck]... [HAVING Bedingung]
\-----/
```

Alle direkt angegebenen Spaltennamen, die hinter SELECT aufgeführt sind, sind auch in der GROUP BY-Klausel aufzuführen. Eine Gruppierung auf Spalten alleine ist normalerweise nicht besonders sinnvoll, da dies auch mit SELECT DISTINCT erreicht werden kann. Im SELECT-Teil können aber auch die zuvor dargestellten Gruppenfunktionen eingesetzt werden, um aussagekräftige Informationen zu gewinnen. Diese können auch in der HAVING-Bedingung bei der Auswahl der zu selektierenden Gruppen eingesetzt werden.

Beispiele:

⇒ Liste aller Abteilungen mit Durchschnitts- sowie Spitzengehalt der Angestellten?

```
SELECT ANR, AVG(Gehalt), MAX(Gehalt)
FROM Pers
GROUP BY ANR
```

⇒ Welche Abteilungen haben mehr als fünf Mitarbeiter?

```
SELECT ANR
FROM Pers
GROUP BY ANR
HAVING COUNT(*) > 5
```

⇒ Für welche Abteilungen zwischen K50 und K60 ist das Durchschnittsgehalt größer als 50000?

```
SELECT ANR
FROM Pers
WHERE ANR BETWEEN 'K50' and 'K60'
GROUP BY ANR
HAVING AVG(Gehalt) > 50000
```

5.3.8 Unterabfragen (Subqueries)

Bisher wurden Bedingungen in der WHERE-Klausel in einer Art formuliert, bei der die Vergleichsoperanden Attribute oder Konstanten waren. Als rechter Vergleichsoperand kann auch das Ergebnis einer SELECT-Anweisung dienen, sofern dieses genau eine Ergebnissezeile besitzt.

Beispiele:

⇒ Welche Piloten (PNR) wohnen am gleichen Wohnort wie der Pilot mit der PNR 'P27'?

```
SELECT P1.PNR
FROM Pilot P1
WHERE P1.WOrt = (SELECT P2.WOrt FROM Pilot P2
                  WHERE P2.PNR = 'P27')
```

⇒ Welche Mitarbeiter (PNR) verdienen das meiste Gehalt?

```
SELECT P.PNR
FROM Pers P
WHERE P.Gehalt = (SELECT MAX(Gehalt) FROM Pers)
```

Häufig werden Unterabfragen mit dem Mengentest IN eingesetzt. Hierbei kann die Unterabfrage auch eine Menge von Ergebnissen liefern, mit denen der Vergleich durchgeführt wird.

Beispiele:

⇒ Welche Piloten (PNR) sind von Stuttgart abgeflogen?

```
SELECT P.PNR
FROM Pilot P
WHERE P.PNR IN (SELECT F.PNR FROM Flug F
                  WHERE F.Start = 'Stuttgart')
```

⇒ Welche Piloten (Name) sind noch nie geflogen?

```
SELECT P.Name
FROM Pilot P
WHERE P.PNR NOT IN (SELECT F.PNR FROM Flug F)
```

Innere und äußere Relation können bei Subqueries identisch sein. Die Schachtelungstiefe kann dabei beliebig tief sein. Join-Berechnungen mit Subqueries, in denen die Unterabfragen sich auf Spalten der Oberabfrage beziehen, werden auch als korrelierende Subqueries bezeichnet. Ohne Bezug spricht man von einfachen Unterabfragen.

Beispiel:

⇒ Korrelierende Abfrage für: Welche Piloten (Name) sind von Stuttgart abgeflogen?

```
SELECT P.Name
FROM Pilot P
WHERE 'Stuttgart' IN (SELECT F.Start FROM Flug F
                      WHERE F.PNR = P.PNR)
```

5.3.8.1 Quantifizierte Bedingungen

Während der Operator IN bei Unterabfragen nur die Prüfung auf Gleichheit zulässt, kann mit den Operatoren ALL, SOME und ANY bei Unterabfragen auch ein Test mit jedem der Standardvergleichsoperatoren durchgeführt werden. Ein Vergleich in Verbindung mit ALL wird in der Weise durchgeführt, dass der zu testende Wert mit jeder Zeile des Subqueryergebnisses verglichen wird. Ist in allen diesen Zeilen die Bedingung erfüllt, gilt der Bedingungsausdruck als wahr. Beim Einsatz von ANY reicht es aus, wenn die Bedingung für mindestens einen Datensatz der Subquery erfüllt ist. SOME hat die gleiche Bedeutung wie ANY.

Beispiel:

⇒ Finden Sie die Vorgesetzten, die mehr verdienen als alle ihre Angestellten.

```
SELECT M.PNR
FROM Pers M
WHERE M.GEHALT > ALL (SELECT P.Gehalt FROM Pers P
                       WHERE P.Vorgesetzter = M.PNR) AND
M.PNR IN (SELECT Vorgesetzter FROM Pers)
```

5.3.8.2 Existenztests

Bei Existenztests wird kein Wert der Unterabfrage ausgewertet. Es wird nur festgestellt, ob überhaupt Werte existieren. Es genügt also, dass die Unterabfrage eine Ergebnistabelle produziert, unabhängig davon, welche Werte in ihr vorkommen. Deshalb wird in der Unterabfrage meist ein SELECT * verwendet. EXISTS wird zu FALSE ausgewertet, wenn die Unterabfrage auf die leere Menge führt.

Beispiele:

⇒ Finden Sie die Vorgesetzten, die mehr verdienen als alle ihre Angestellten.

```
SELECT M.PNR
FROM Pers M
WHERE NOT EXISTS (SELECT * FROM Pers P
                   WHERE P.Vorgesetzter = M.PNR AND
                   P.Gehalt > M.Gehalt) AND
M.PNR IN (SELECT Vorgesetzter FROM Pers)
```

⇒ Finden Sie die Namen der Piloten, die mindestens einmal geflogen sind.

```
SELECT P.Name
FROM Pilot P
WHERE EXISTS (SELECT * FROM Flug F
              WHERE F.PNR = P.PNR)
```

Mit EXISTS können alle mit ALL oder ANY definierten Abfragen auch formuliert werden:

- $X \theta \text{ ANY } (\text{SELECT } Y \text{ FROM } T \text{ WHERE } P) \equiv$
 $\text{EXISTS } (\text{SELECT } * \text{ FROM } T \text{ WHERE } (P) \text{ AND } (X \theta (T.Y)))$
- $X \theta \text{ ALL } (\text{SELECT } Y \text{ FROM } T \text{ WHERE } P) \equiv$
 $\text{NOT EXISTS } (\text{SELECT } * \text{ FROM } T \text{ WHERE } (P) \text{ AND NOT}(X \theta (T.Y)))$

5.3.8.3 Unterabfragen in der FROM-Klausel

In der FROM-Klausel werden die Relationen angegeben, auf die in der Abfrage zugegriffen werden soll. Da das Ergebnis einer SELECT-Anfrage eine Relation darstellt, ist es naheliegend, dass man innerhalb der FROM-Klausel anstatt von Relationen auch ein SELECT angeben kann. Das Select-Ergebnis wird dann als Tabelle für die Gesamtanfrage verwendet. Im Beispiel:

Beispiel:

- ⇒ Wie groß ist die Abweichung des Gehalts der Mitarbeiter zum Durchschnittsgehalt ihrer Abteilung?

```
SELECT M.PNR, M.Gehalt - A.AVGGehalt
FROM Pers M, (SELECT ANR, AVG(Gehalt) as AVGGehalt FROM Pers GROUP BY
ANR) AS A
WHERE M.ANR = A.ANR
```

5.3.9 NULL-Werte

Für jedes Attribut kann festgelegt werden, ob NULL-Werte zugelassen sind oder nicht. NULL kann dabei unterschiedliche Bedeutungen haben:

- Datenwert ist momentan nicht bekannt.
- Attributwert existiert nicht für ein Tupel.

Wird nun ein NULL-Wert arithmetisch mit anderen Werten verrechnet, so ist das Ergebnis wieder unbekannt und damit NULL. Wird ein Vergleich mit einem NULL-Wert durchgeführt, so ist das Ergebnis weder falsch noch wahr, sondern unbekannt. SQL arbeitet daher mit einer dreiwertigen Logik, die den zusätzlichen Wahrheitswert UNKNOWN (?) enthält. Dies führt zu einer Logik mit Operationen gemäß folgender Tabellen:

NOT		AND				OR			
T	F	T	T	F	?	T	T	F	?
F	T	F	F	F	F	F	T	?	?
?	?	?	?	F	?	?	T	?	?

Das Ergebnis ? bei der Auswertung einer WHERE-Klausel wird dabei wie FALSE behandelt. Lässt man das Ergebnis eines Vergleichs mit einem NULL-Wert ausgeben, wird der Wert Unknown meist auch als NULL für einen unbekannten Wert dargestellt.

Zum Abprüfen von NULL-Werten gibt es ein spezielles Prädikat:

/ IS [NOT] NULL \

Beispiel:

```
⇒ SELECT PNR, Name FROM Pers
    WHERE Gehalt IS NULL
```

Die dreiwertige Logik führt auch zu unerwarteten Ergebnissen:

WHERE (Attr = NULL) OR (Attr <> NULL) ergibt immer ein leeres Ergebnis. NULL-Werte werden zudem bei SUM, AVG, MIN und MAX nicht berücksichtigt, während COUNT(*) alle Tupel zählt.

5.3.10 Mengentheoretische Funktionen

Die mengentheoretischen Operatoren erlauben Ergebnisse von Datenbankanfragen mengentheoretisch zu verknüpfen. Dies verlangt jedoch die Vereinigungsverträglichkeit der Ergebnisse, d.h. die Anzahl der Spalten sowie die Datentypen der selektierten Spalten der zu verknüpfenden Anfragen müssen gleich sein. Als Operatoren stehen dabei zur Verfügung:

UNION [ALL]: Vereinigung der Ergebnistabellen, wobei Duplikate gelöscht werden. Dies kann durch die Option ALL verhindert werden.

INTERSECT: Mengendurchschnitt der Ergebnistabellen.

EXCEPT | MINUS: Mengendifferenz der Ergebnistabellen.

Diese Operationen sind vor allen dann nützlich, wenn gemeinsam zu betrachtende Daten in verschiedenen Tabellen/Spalten auftreten. So können einzelne Anfragen auf die benötigten Spalten der einen Tabelle zugreifen, weitere Anfragen führen Zugriffe auf die anderen Tabellen/Spalten aus. Die einzelnen Anfragen können dann mit den Mengenoperationen verknüpft werden.

Beispiel:

⇒ Welche Abteilungen (ANR) haben ein Budget größer 1Mio Euro oder einen Mitarbeiter, der mehr als 100000 Euro verdient.

```
SELECT ANR FROM Abteilung WHERE Budget > 1000000
UNION
SELECT ANR FROM Pers WHERE Gehalt > 100000
```

5.3.11 Begrenzen der Zeilenanzahl

Möchte man bei einer Anfrage nicht das vollständige Anfrageergebnis dargestellt bekommen, ist eine Limitierung der darzustellenden Tupel nützlich. Dies unterstützen die meisten Datenbankverwaltungssysteme. Da dies jedoch nicht standardisiert ist, unterscheiden sich die verschiedenen DBVS bei dieser Funktionalität erheblich. So setzt MySQL eine Limit-Klausel ein, DB2 und PostgreSQL verwendet eine FETCH-Klausel. Beim Microsoft SQL-Server gibt es hierfür die TOP-Klausel, Oracle verwendet eine WHERE-Bedingung über die Zeilennummer. Hier ist damit bei Bedarf beim jeweiligen DBVS nachzulesen, wie diese Funktionalität umgesetzt werden kann. In der Anwendungsentwicklung mit der Verwendung von SQL innerhalb einer Programmiersprache hat diese Funktionalität eine untergeordnete Bedeutung und wird hier daher nicht detaillierter betrachtet.

5.3.12 Setzen von Änderungssperren

Das Setzen von Sperren zur Verhinderung von Mehrbenutzeranomalien durch parallelen Zugriff übernimmt das Datenbanksystem. Dabei wird das parallele Lesen von Datensätzen zugelassen. Wenn ein Datensatz selektiert wird, der im weiteren Transaktionsverlauf geändert werden soll, kann dies störend sein. Mit der Option FOR UPDATE am Ende eines SELECTs wird das Datenbanksystem veranlasst, eine Schreibsperrre auf die selektierten Datensätze zu setzen, so dass parallele Zugriffe auf die Ergebniszellen nicht mehr möglich sind. Genauere Informationen hierzu finden sich in Kapitel 9.

5.3.13 Relationale Division mit SQL

Der Divisionsoperator der Relationenalgebra findet in SQL keine direkte Umsetzung. Zur Erinnerung: Die Division ist für folgende Fragestellungen geeignet:

- ▶ Ermittlung der Lieferanten, die *alle* roten Teile liefern.
- ▶ Ermittlung der Studenten, die alle Vorlesungen besucht haben.
- ▶ Ermittlung der Kunden, die schon alle Produkte einer Produktlinie ausprobiert haben.

Der entscheidende Punkt bei diesen Fragestellungen ist, dass Objekte gesucht werden, die eine Beziehung zu allen Objekten der Bezugsmenge haben. Ein Beispiel:

A	B
a1	b1
a1	b2
a1	b3
a2	b1
a2	b3
a3	b2
a3	b3
a3	b4
a4	b1

$$\div \begin{array}{|c|} \hline B \\ \hline b2 \\ \hline b3 \\ \hline \end{array} = \begin{array}{|c|} \hline A \\ \hline a1 \\ \hline a3 \\ \hline \end{array}$$

Die Tabellen in diesem Beispiel seien mit T1, T2 und T3 benannt, damit gilt $T3 = T1 \div T2$.

Eine häufig angegebene Umsetzung des Divisionsoperators mit SQL ist wie folgt aufgebaut:

```
SELECT DISTINCT x.A
  FROM T1 x
 WHERE NOT EXISTS
   (SELECT * FROM T2 y
    WHERE NOT EXISTS
      (SELECT * FROM T1 z
       WHERE (z.A = x.A) AND (z.B = y.B)))
```

Auch wenn diese Lösung in der Datenbankliteratur allgemein akzeptiert ist, gibt es eine Variante, die meist verständlicher erscheint:

```
SELECT DISTINCT A
  FROM T1
 WHERE B IN (SELECT B FROM T2)
 GROUP BY A
 HAVING COUNT(*) = (SELECT COUNT(*) FROM T2)
```

Insbesondere haben Untersuchungen gezeigt, dass diese Variante auch bei einigen Systemen performanter ist. Ein Problem entsteht jedoch, wenn T2 leer ist, dann erzeugen die beiden Abfragen unterschiedliche Ergebnisse. Die zweite Formulierung erzeugt die leere Menge, während die erste Lösung alle T1-Elemente auffüllt.

5.4 Rekursion in SQL3

Eine Beschränkung der Relationenalgebra und den bisher dargestellten SQL-Möglichkeiten war die fehlende Berechnung der transitiven Hülle. In SQL3 wurde die Formulierung derartiger Anfragen standardisiert, IBM DB2 hat dies in seinen SQL-Dialekt aufgenommen, ORACLE hat für dieses Problem eine eigene Syntax entwickelt. Zur Darstellung wird exemplarisch eine Personaltabelle mit folgendem Aufbau eingeführt:

```
PERS (Name, Gehalt, Vorgesetzter REFERENCES PERS)
```

Will man die Namen und Gehälter aller Angestellten bestimmen, deren direkter Vorgesetzter Peters heißt und deren Gehalt größer als 100000 ist, so reicht dazu folgende Anfrage:

Beispiel:

```
⇒ SELECT NAME, GEHALT
  FROM PERS
 WHERE Vorgesetzter = 'Peters' AND Gehalt > 100000
```

Komplizierter wird es, falls sämtliche Angestellten gewünscht sind, die mehr als 100000 verdienen und in der Unternehmenshierarchie unterhalb (direkt oder über mehrere Stufen) von Peters arbeiten. Diese Art Anfrage ist mit den bisherigen Sprachmitteln nicht formulierbar, man benötigt Rekursion. Dabei ist zunächst mit einer WITH-Klausel eine Sicht zu definieren, die hier UNTERGEBENE genannt wird. Diese Sicht hat zwei Teile, die mit UNION zu verbinden sind. Der erste Teil ist eine gewöhnliche, nicht-rekursive Unterabfrage, die in diesem Beispiel die Namen und Gehälter aller Angestellten findet, deren unmittelbarer Vorgesetzter Peters ist. Der zweite Teil ist der rekursive Teil, die zu der Sicht weitere Tupel hinzufügt, wobei auf die bereits vorhandenen Tupel Bezug genommen wird. In dem Beispiel werden der Sicht UNTERGEBENE solche Angestellte hinzugefügt, die Angestellte als Vorgesetzte haben, die sich bereits in UNTERGEBENE befinden. Diese Berechnung wird solange iteriert, bis sich der Inhalt des Views nicht mehr ändert. Danach kann die Sicht wie eine gewöhnliche Tabelle im Rahmen eines SELECT-Ausdrucks angefragt werden.

Beispiel:

```
⇒ WITH UNTERGEBENE (NAME, GEHALT) AS
  (( SELECT NAME, GEHALT
    FROM PERS
    WHERE Vorgesetzter = 'Peters' )
  UNION
  ( SELECT P.NAME, P.GEHALT
    FROM UNTERGEBENE U, PERS P
    WHERE P.Vorgesetzter = U.NAME ))
  SELECT NAME
  FROM UNTERGEBENE
  WHERE GEHALT > 100000
```

Bei der Formulierung der Rekursion muss man vorsichtig bei der Formulierung der Rekursion sein, da leicht Endlos-Schleifen entstehen können, wodurch die Anfrage nicht durchgeführt werden kann.

5.5 Datenmanipulationen

5.5.1 Zeilen-INSERT

Das Einspeichern von neuen Daten erfolgt mit dem Befehl INSERT, der die Daten zeilenweise in jeweils eine Tabelle einträgt. Bevor dabei eine Zeile in der Datenbank gespeichert wird, werden die

Integritätsbedingungen der Tabelle geprüft. Fallen diese Prüfungen negativ aus, erhält man eine Fehlermeldung.

```
/-----\
INSERT INTO {Tabellenname | Viewname }
[ (Spaltenname [,Spaltenname] ... )]
VALUES (Wert [,Wert] ... )
\-----/
```

Es kann immer nur in eine Tabelle eine Zeile eingefügt werden. Alternativ kann auch in eine View ein Datensatz eingefügt werden (siehe hierzu Abschnitt 5.9.1). Dies ist aber nur dann möglich, wenn man die Daten direkt auf eine Tabelle übertragen kann und in der Tabelle dann alle Integritätsbedingungen erfüllt sind. So müssen beispielsweise alle NOT NULL-Werte der Tabelle in der View enthalten sein. Weiterhin dürfen in der View keine Gruppierungen oder Berechnungen innerhalb der Einfügespalten enthalten sein.

Nach der Angabe des Tabellen- oder Viewnamens kann optional eine Spaltenliste folgen. Diese benennt die Spalten, in die Werte eingesetzt werden sollen sowie die Reihenfolge, in der die Spaltenwerte angegeben werden. In die nicht aufgeführten Spalten werden NULL-Werte eingesetzt. Die Reihenfolge der Spalten muss hier nicht der Reihenfolge der Tabellendefinition entsprechen.

Fehlt die Spaltenliste, erwartet das Datenbanksystem Werte für jedes Attribut der Tabelle in genau der Reihenfolge der Tabellendefinition. Diese Variante sollte nur in Ausnahmefällen verwendet werden, da Tabellendefinitionen vielfach eine Dynamik besitzen. Änderungen an der Tabelle führen hier immer zu einer Anpassung der INSERT-Befehle. Bei der Spaltenangabe müssen nur die Befehle angepasst werden, die direkt von der Tabellenänderung betroffen sind.

Die zu speichernden Werte werden ebenfalls in einer Liste von Konstanten angegeben. Die Datentypen der Konstanten müssen dabei zu den Datentypen der Spalten passen, wobei teilweise vom Datenbanksystem automatisch Konvertierungen vorgenommen werden, so dass beispielsweise ein Datum als Zeichenkette angegeben werden kann.

Beispiele:

```
⇒ INSERT INTO Pilot
    VALUES ('A1736732', 'Hannahmax',
            NULL, NULL, '12.12.2002')
```

```
⇒ INSERT INTO Pilot (PNR, Name, UTermin)
    VALUES ('A1736732', 'Hannahmax', '12.12.2002')
```

```
⇒ INSERT INTO Pilot (PNR, UTermin)
    VALUES ('A1736732', '12.12.2002')
```

Diese Operation wird zurückgewiesen, das Einfügen eines Piloten erfordert die Angabe eines Namens.

5.5.2 Mengen-INSERT

Für das Kopieren von Daten von einer Tabelle in eine andere gibt es eine besondere INSERT-Variante:

```
/-----\
INSERT INTO {Tabellenname | Viewname }
[ (Spaltenname [,Spaltenname] ... )]
SELECT-Anweisung
\-----/
```

Beispiele:

```

⇒ INSERT INTO Pers (PNR, Name)
    SELECT PNR, Name FROM Pilot
    Die Pilotdaten werden in eine Personaltabelle kopiert.

⇒ INSERT INTO FlugBackup SELECT * FROM Flug
    Die Flugdaten werden in eine Backup-Tabelle gesichert. Dies ist sinnvoll, wenn
    man größere Änderungen an einer Tabelle vornehmen möchte und so im Fehlerfall
    einfach den Originalzustand wieder herstellen kann, indem die gesicherten Daten
    wieder zurückgespielt werden.

```

5.5.3 DELETE

Das Löschen von Datensätzen ist mengenorientiert. Dabei ist beim DELETE-Befehl anzugeben, aus welcher Tabelle Datensätze und, mit Hilfe einer optionalen WHERE-Bedingung, welche Datensätze aus der Tabelle entfernt werden sollen. Die Möglichkeiten bei der Formulierung der WHERE-Bedingung entsprechen denen von SELECT.

```

/-----\
DELETE FROM {Tabellenname | Viewname } [Aliasname]
    [WHERE Bedingung]
\-----/

```

Beispiele:

```

⇒ Löchen Sie den Piloten mit der PNR 13813.
    DELETE FROM Pers
        WHERE PNR = '13813'.

⇒ Löchen Sie alle Piloten, die nie geflogen sind.
    DELETE FROM Pilot P
        WHERE NOT EXISTS (SELECT * FROM Flug F WHERE
            F.PNR = P.PNR

```

5.5.4 UPDATE

Das Ändern von Daten in einzelnen Attributen erfolgt mit dem Befehl UPDATE. Ein UPDATE kann dabei Änderungen an einer Menge von Datensätzen in einer Tabelle durchführen. Betroffen sind alle Datensätze, die eine WHERE-Bedingung erfüllen. Die Möglichkeiten bei der Formulierung der WHERE-Bedingung entsprechen denen von SELECT.

```

/-----\
UPDATE {Tabellenname | Viewname } [Aliasname]
    SET (Spaltenname = Wert,
        [, Spaltenname = Wert] ... )
    [WHERE Bedingung]
\-----/

```

Beispiel:

```

⇒ Geben Sie allen Personen der Abteilung 'Forschung' eine 5-prozentige Gehalts-
erhöhung.
    UPDATE Pers P
        SET Gehalt = Gehalt * 1.05
        WHERE EXISTS (SELECT * FROM Abteilung A
            WHERE A.ANR = P.ANR AND A.Name = 'Forschung')

```

5.5.5 TRUNCATE

Für das schnelle Löschen aller Einträge innerhalb einer Tabelle dient der Befehl TRUNCATE, der es zusätzlich erlaubt, den von der Tabelle verwendeten Speicher für andere freizugeben (DROP STORAGE). TRUNCATE läuft nicht innerhalb einer Transaktion ab, ein ROLLBACK zur Wiederherstellung der Daten ist nicht möglich.

```
/-----\
TRUNCATE TABLE Tabellenname [{DROP | REUSE} STORAGE]
\-----/
```

5.6 Transaktionsbefehle

Transaktionen sind Arbeitseinheiten, die als ganzes oder gar nicht auszuführen sind. Tritt innerhalb einer Transaktion ein Problem auf, wird die gesamte Arbeitseinheit abgebrochen. So wird beispielsweise bei einer Banküberweisung, die Änderungen an mindestens zwei Kontoständen zur Folge hat, erreicht, dass nie nur ein Kontostand verändert wird. Tritt ein Fehler nach der ersten Änderung auf, wird diese zurückgenommen. Nur wenn beide Kontenänderungen durchgeführt sind, bleiben die Änderungen erhalten.

Jede Transaktion hat einen Anfang und ein Ende. Der Start einer Transaktion wird meist durch den ersten DML-Befehl nach dem Anmelden oder dem Abschluss einer vorhergehenden Transaktion gestartet.

5.6.1 COMMIT

COMMIT beendet eine Transaktion erfolgreich, die in der Transaktion durchgeföhrten Änderungen werden dauerhaft in die Datenbank übernommen.

```
/-----\
COMMIT [WORK]
\-----/
```

Der Anhang WORK ist im Standard vorgesehen, bei vielen DBS ist die Angabe optional. Die Auswirkungen sind bei beiden Varianten gleich.

5.6.2 ROLLBACK

ROLLBACK bewirkt, dass alle Änderungen, die in der aktuellen Transaktion durchgeführt wurden, zurückgenommen werden. Der Datenbankzustand entspricht dem Zustand zu Beginn der Transaktion.

```
/-----\
ROLLBACK [WORK]
[TO [SAVEPOINT] savepoint]
\-----/
```

5.6.3 Rücksetzpunkte

Wesentlich für die große Akzeptanz des Transaktionskonzepts sind unter anderem seine einfache Benutzerschnittstelle, seine klare Fehlersemantik (Alles-oder-Nichts), die Nutzbarkeit in zentralisierten und verteilten Systemen sowie die Verfügbarkeit effizienter Implementierungen. Andererseits ist die Einfachheit des ACID-Paradigmas, welches von flachen Transaktionen ohne Binnenstruktur ausgeht, für nicht wenige Anwendungsfälle zu restriktiv. Diese Beschränkungen wurden frühzeitig erkannt, eine kaum überschaubare Anzahl von Erweiterungen des ACID-Konzepts wurden vorgeschlagen. Als am bedeutsamsten haben sich dabei verschiedene Varianten geschachtelter Transaktionen herausgestellt, welche eine Zerlegung von Transaktionen in interne Sub-Transaktionen unterstützen.

5.6.3.1 Beschränkungen des ACID-Konzepts

Eine Hauptbeschränkung des ACID-Konzepts ist, dass es vor allem auf kurze Transaktionen ausgerichtet ist, die nur relativ wenige Datenbankobjekte bearbeiten und deren Bearbeitungszeit im Sekunden oder maximal im Minutenbereich liegt. Solche kurzen Transaktionen dominieren in vielen Bereichen, insbesondere innerhalb von OLTP-Anwendungen etwa bei Banken oder Reservierungssystemen. Transaktionen wie Kontostandsabfragen, Geldabhebungen, Flugbuchungen usw. sind in sehr großer Anzahl im Dialog auszuführen und damit sehr zeitkritisch.

Bei komplexeren Verarbeitungsvorgängen mit sehr vielen Objektreferenzen und längerer Ausführungszeit entsteht jedoch eine Reihe von Problemen, insbesondere aufgrund der Eigenschaften A (Atomarität) und I (Isolation) des ACID-Konzepts. Ein Fehler während der Ausführung verlangt aufgrund der Alles-oder-Nichts-Eigenschaft das vollständige Rücksetzen der Transaktion. Der Arbeitsverlust ist jedoch ungleich höher als bei einer kurzen Transaktion und vielfach nicht akzeptabel. Dieses Problem wird dadurch verschärft, dass die Wahrscheinlichkeit eines Transaktions-, System- oder Gerätefehlers proportional zur Transaktionsdauer zunimmt.

Isolationsprobleme ergeben sich für langlebige Transaktionen dadurch, dass die Wahrscheinlichkeit von Synchronisationskonflikten im Allgemeinen quadratisch mit der Transaktionsdauer ansteigt, da einerseits die Anzahl der Objektsperren pro Transaktion und andererseits die Sperrdauer proportional zur Transaktionslänge steigen (siehe hierzu Kapitel 9). Die zur Sicherstellung der Serialisierbarkeit erforderliche strikte Isolation führt somit bei langen Transaktionen oft zu inakzeptablen Leistungseinbußen. Synchronisationsverfahren, welche primär Rücksetzungen zur Auflösung von Synchronisationskonflikten verwenden (beispielsweise optimistische Verfahren, siehe hierzu Abschnitt 9.6), sind aufgrund des zu befürchtenden extremen Arbeitsverlusts als noch ungeeigneter einzustufen.

Diese und weitere Probleme zeigen sich beispielsweise in folgenden Bereichen mit länger andauernden Verarbeitungsvorgängen:

- ▶ Vielfach erfolgen Massenänderungen des Datenbestands innerhalb von Batch-Transaktionen, welche zu Zeiten eines geringen Dialogbetriebs (meist nachts) durchgeführt werden. Als Beispiel diene etwa die Berechnung und Gutschreibung der Zinsen für alle Sparkonten einer Bank, was durchaus eine Bearbeitungszeit von mehreren Stunden erfordern kann. Werden alle Änderungen in einer Transaktion durchgeführt, führt beispielsweise ein Rechnerausfall während der Bearbeitung zum vollständigen Zurücksetzen aller bereits erfolgter Kontoänderungen. Die gesamte Berechnung müsste also erneut vorgenommen werden.
Dieses Problem wird offenbar umgangen, wenn jede Kontoänderung in einer eigenen Transaktion erfolgt. Hierbei besteht jedoch nach einem Rechnerausfall das Problem herauszufinden, welche Konten bereits bearbeitet wurden und für welche die Zinsberechnung erneut veranlasst werden muss. Dies führt zu einer komplexen Fehlerbehandlung, die manuelle Eingriffe des Systemverwalters verlangt.

- ▶ Probleme in OLTP-Anwendungen verursachen sogenannte Mehrschritt-Transaktionen, welche mehrere Dialogschritte zwischen Benutzer und Datenbanksystem umfassen. So kann beispielsweise für eine Flugbuchung in einem ersten Schritt die aktuelle Sitzbelegung eines Flugzeugs erfragt werden um danach in einem zweiten Schritt einen Platz zu reservieren. Die Menge der Dialogschritte bildet aus Anwendungssicht jeweils eine logisch zusammenhängende Einheit. Ihre Ausführung innerhalb einer Transaktion impliziert jedoch hohe Bearbeitungszeiten, da aufgrund der Benutzerinteraktion (Denkzeiten) unbestimmt lange Verzögerungen auftreten können. Dies führt bei der Verwendung langer Sperren zu meist inakzeptablen Behinderungen für andere Benutzer (Blockierung aller Reservierungen eines bestimmten Flugs).
- ▶ Eine Verallgemeinerung zu Mehrschritt-Transaktionen sind langlebige Verarbeitungsvorgänge mit zahlreichen Teilschritten, wie sie im Rahmen von Workflow-Management-Systemen zur Unterstützung von Geschäftsprozessen auftreten. So könnte für eine Reiseplanung neben der Flugreservierung auch die Buchung von Hotel und Leihwagen in einem Vorgang durchgeführt werden. Bei solchen Vorgängen ist ein Zurücksetzen von bereits erfolgten Änderungen früherer Teilschritte meist nicht akzeptabel. Wenn beispielsweise eine Hotelreservierung aufgrund eines Rechnerausfalls scheitert, wäre es sicher unerwünscht, dass eine zuvor erfolgte Flugreservierung auch zurückgesetzt wird. Anstelle des Zurücksetzens des gesamten Vorgangs im Rahmen einer Backward-Recovery ist eine Forward-Recovery zu unterstützen, welche auf den erfolgreichen Teilschritten aufbaut und die Verarbeitung nach vorne fortsetzt.
- ▶ Besonders lange Verarbeitungsvorgänge bestehen in datenbankgestützten Entwurfsanwendungen wie CAD- und CASE. Der Entwurf beispielsweise eines neuen Maschinenteils oder Software-Pakets dauert häufig Wochen und Monate. Die Durchführung solcher Vorgänge als ACID-Transaktionen scheidet schon deshalb aus, da ansonsten im Fehlerfall ein inakzeptabler Arbeitsverlust entstehen würde. Außerdem ist die strikte Isolation herkömmlicher Synchronisationsverfahren zu restriktiv, da Entwurfsvorgänge häufig kooperativ im Team bearbeitet werden. Es sind daher Mechanismen der Kooperation innerhalb von Entwurfsvorgängen bereitzustellen. Weitere Anforderungen an das Verarbeitungskonzept ergeben sich aus der Notwendigkeit, unterschiedliche Versionen der Entwurfsobjekte zu verwalten.

Die genannten Beschränkungen sind zum Teil eine Folge der fehlenden Binnenstruktur von ACID-Transaktionen. Diese werden auch als flache Transaktionen bezeichnet, bestehend aus einer Folge von Datenbankoperationen. Eine weitere Strukturierung wird nicht vorgenommen, hinsichtlich Atomarität und Synchronisation wird nur die Transaktion als Ganzes betrachtet. Voraussetzung zur Problembehebung ist somit die Einführung einer Binnenstruktur, so dass eine Transaktion bzw. ein Verarbeitungsvorgang intern in kleinere Ausführungseinheiten zerlegt wird, für die besondere Eigenschaften gewährleistet werden. Dies führt zu der Idee der Transaktionen mit Rücksetzpunkten.

5.6.3.2 Transaktionen mit Rücksetzpunkten

Ein erster Schritt zur Reduzierung des Arbeitsverlusts im Fehlerfall ist die Einführung von Rücksetzpunkten oder Savepoints innerhalb von Transaktionen. Sie stellen eine Abschwächung der Atomaritätszusicherung dar und ermöglichen das partielle Zurücksetzen von Transaktionen durch Zurückgehen auf einen Rücksetzpunkt. Dieser Ansatz wird inzwischen von vielen Datenbanksystemen unterstützt, er ist auch im SQL3-Standard vorgesehen.

```
/—————\  
SAVEPOINT savepoint  
—————/
```

Die SAVEPOINT-Operation bewirkt das Setzen eines Rücksetzpunkts und damit die Sicherung des erreichten Transaktionszustands. Dies betrifft die bis dahin durchgeföhrten Datenbankänderungen sowie Informationen über die vorliegenden Sperren der Transaktion und Cursor-Positionen. Als Ergebnis der SAVE-Operation erhält die Anwendung eine eindeutige Nummer für den Rücksetzpunkt, wobei diese Nummern streng monoton wachsend vergeben werden. Im Fehlerfall kann mit dem Befehl ROLLBACK auf einen bestimmten Rücksetzpunkt R zurückgegangen werden. Dies impliziert ein partielles UNDO, bei dem alle nach R erfolgten Änderungen in umgekehrter Reihenfolge zurückgesetzt werden. Die Änderungen, die vor R ausgeführt wurden, haben Bestand und müssen nicht wiederholt werden.

Für diese Art von Rücksetzpunkten erfolgt die Sicherung des Zwischenzustands einer Transaktion nur auf Seite des Datenbanksystems, nicht jedoch auf Anwendungsseite. Daher bieten sie auch nur eine Unterstützung hinsichtlich Transaktionsfehler, nicht jedoch gegenüber Systemfehlern. Denn nach einem Rechnerausfall könnte das Datenbanksystem eine Rücksetzung auf den zuletzt erreichten Rücksetzpunkt einer betroffenen Transaktion vornehmen. Jedoch wäre keinerlei Information über den zugehörigen Status des jeweiligen Anwendungsprogramms verfügbar, so dass keine Fortführung der Transaktion ausgehend vom Rücksetzpunkt möglich ist.

Zur Reduzierung des Arbeitsverlusts gegenüber Rechnerausfällen sind somit persistente Rücksetzpunkte erforderlich, welche eine koordinierte Sicherung für Anwendungs- und Datenbankobjekte gewährleisten. Insbesondere müsste dazu neben dem Datenbanksystem auch das Laufzeitsystem für die Anwendungsprogramme als transaktionsgeschützter Ressourcen-Manager realisiert sein, der mit dem Datenbanksystem ein gemeinsames Commit-Protokoll sowie koordinierte Rücksetzpunkte durchführt. Im Fehlerfall wird dann die Verarbeitung auf den letzten persistenten Savepoint zurückgesetzt und von dort über eine Forward-Recovery zum erfolgreichen Abschluss gebracht. Damit könnte beispielsweise auch das zuvor geschilderte Wiederanlaufproblem für Batch-Transaktionen gelöst werden. Derzeitige Systeme unterstützen jedoch meist noch keine persistenten Savepoints.

5.6.4 LOCK TABLE

Das Sperren einer ganzen Tabelle innerhalb einer Transaktion kann mit dem Befehl LOCK TABLE erfolgen (genauere Informationen hierzu in Kapitel 9):

```
/-----\
LOCK TABLE Tabellename
  IN {ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE |
       SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE } MODE
\-----/
```

5.6.5 SET TRANSACTION

Zur Verringerung der Maßnahmen zum Ausschluss von Schreib-Lese-Konflikten mit anderen parallel ablaufenden Transaktionen kann man die aktuelle Transaktion, wenn diese nur Daten aus der Datenbank ausliest, bei ORACLE in den READ-ONLY-Modus versetzen. Die Option ISOLATION LEVEL bestimmt, inwieweit das Datenbanksystem Sperren für den Mehrbenutzerbetrieb setzen soll und damit welche Mehrbenutzeranomalien auftreten können (siehe hierzu Kapitel 9).

```
/-----\
SET TRANSACTION
  [READ ONLY]
  [ISOLATION LEVEL
    {READ UNCOMMITTED | READ COMMITED |
     REPEATABLE READ | SERIALIZABLE }]
\-----/
```

5.7 Temporäre Tabellen

Zum Abspeichern und zum Weitergeben von Zwischenergebnissen von einer Transaktion zur anderen können temporäre Tabellen eingesetzt werden, die nur kurzzeitig im Datenbanksystem bestehen bleiben. Eine temporäre Tabelle unterscheidet sich von Standardtabellen durch:

- ▶ Zum Zeitpunkt des ersten Zugriffs aus dem Anwendungsprogramm oder der SQL-Sitzung ist die Tabelle immer leer.
- ▶ Die Tabelle kann bei einem Commit automatisch vom DBS geleert werden.
- ▶ Die Tabelle wird spätestens am Ende der Datenbanksitzung gelöscht.
- ▶ Temporäre Tabellen können verändert werden, auch wenn die Transaktion im READ-ONLY-Modus abläuft.

Der Standard unterscheidet drei Arten von temporären Tabellen: **declared local temporary table**, **created local temporary table** und **global temporary table**.

Deklarierte Temporäre Tabellen werden mit einer DECLARE-Anweisung definiert und nicht erzeugt. Diese Art der temporären Tabellen ist deshalb nur innerhalb von Programmodulen zulässig und nur dort verwendbar (Vgl. Skript Datenbankprogrammierung). Sie existiert nur zur Programmlaufzeit.

```
/—————\  
DECLARE LOCAL TEMPORARY TABLE Tabelle  
  (Elementliste)  
  [ ON COMMIT { PRESERVE | DELETE } ROWS ]  
—————/
```

Die **ON COMMIT**-Klausel gibt an, ob die Tabelle bei jedem Commit automatisch geleert (**DELETE**) oder unverändert gelassen wird (**PRESERVE**). Der Default ist **DELETE**.

Die beiden anderen Arten von temporären Tabellen werden mit CREATE TABLE erzeugt. Die Tabelle wird jedoch nicht direkt erzeugt, sondern erst beim ersten Zugriff auf diese Tabelle. Dies erfolgt bei lokalen temporären Tabellen transaktionsbezogen und bei globalen sitzungsbezogen. Dies hat auch zur Folge, dass jede Transaktion bzw. Sitzung ihre eigene Version dieser temporären Tabellen hat.

```
/—————\  
CREATE { LOCAL | GLOBAL } TEMPORARY TABLE Tabelle  
  (Elementliste)  
  [ ON COMMIT { PRESERVE | DELETE } ROWS ]  
—————/
```

5.8 Systemkatalog

Zu jeder relationalen Datenbank gehört ein Systemkatalog, der auch als Data Dictionary bezeichnet wird. Er beinhaltet Informationen zu den Komponenten der Datenbank wie Tabellendefinitionen, gespeicherter SQL-Code, Benutzerstatistiken, Datenbankprozessen, Datenbankwachstum und Statistiken zum Leistungsverhalten.

Die Tabellen des Systemkatalogs werden beim Erstellen der Datenbank angelegt und vom Datenbanksystem größtenteils automatisch aktualisiert. Ein CREATE TABLE bewirkt beispielsweise

immer, dass die angelegte Tabelle einschließlich Attribut-, Größen- und Privilegieninformationen auch im Systemkatalog aufgeführt wird.

Der Systemkatalog gehört zu den unentbehrlichen Werkzeugen einer Datenbank. Vergleichbar mit einer Inventurliste eines Handelsbetriebs erlaubt der Systemkatalog Informationen über die Bestandteile der Datenbank zu erfragen. Damit kann beispielsweise geprüft werden, ob ein Objekt in der Datenbank vorhanden ist, ob der Plattenplatz für die Tabelle noch ausreicht oder ob Privilegien für notwendige Zugriffe eingetragen sind.

Administratoren, Entwickler sowie auch der Endbenutzer greifen regelmäßig, wenn auch nicht immer bewusst, auf den Systemkatalog zu. Bei Endbenutzern geschieht dies indirekt. Wenn sich ein Benutzer bei der Datenbank anmeldet, werden beispielsweise bei ORACLE anhand des Systemkatalogs der Benutzername, das Kennwort und die Rechte für die Verbindung verifiziert.

Modellierer, Entwickler und Anwendungsverwalter arbeiten direkt mit dem Systemkatalog, um den Entwicklungsprozess zu verwalten und die existierenden Projekte zu warten. Dies geschieht meist über CASE-Werkzeuge, die alle Informationen aus dem Systemkatalog darstellen.

Datenbankadministratoren (DBAs) nutzen den Systemkatalog sehr intensiv. Der Zugriff geschieht direkt über SQL-Anfragen auf die Tabellen des Systemkatalogs. Teilweise werden die Arbeiten auch durch Werkzeuge erleichtert, die dies für den Administrator übernehmen. Der DBA nutzt die so gewonnenen Informationen zur Benutzerverwaltung und zur Abstimmung der Datenbank für einen optimalen Betrieb.

Die Tabellennamen des Systemkatalogs sind datenbankspezifisch. Beispiele für Katalogtabellen unter ORACLE sind:

DICT: Liste aller Systemkatalogtabellen mit Beschreibung

ALL_TABLES: Alle Tabellen, auf die man zugreifen darf¹⁰

USER_OBJECTS: Alle Objekte im Schema des angemeldeten Benutzers

USER_SYS_PRIVS: Systemprivilegien des angemeldeten Benutzers

USER_TAB_PRIVS: Alle vergebenen und bekommenen Tabellenprivilegien

USER_TAB_PRIVS_MADE: Alle vergebenen Tabellenprivilegien

USER_TAB_PRIVS_REC'D: Alle bekommenen Tabellenprivilegien

ALL_USERS: Alle am System registrierten Benutzer

USER_COMMENTS: Tabellenkommentare

USER_COL_COMMENTS: Spaltenkommentare

V\$SESSION: Alle aktuellen Datenbanksessions¹¹

V\$PARAMETERS: Systemeinstellungen

SESSION_PRIVS: Liste aller möglichen Objektprivilegien

¹⁰Katalogtabellen, die zur Abfrage von Objektinformationen dienen, haben unter ORACLE die Präfixe 'ALL', 'USER' und 'DBA'. Der Präfix 'USER' dient zur Darstellung aller Objekte, die zu dem angemeldeten Benutzer gehören. 'ALL' listet alle Objekte, auf die man zugreifen darf, 'DBA' beinhaltet alle Objekte der Datenbank.

¹¹Systemtabellen, die den aktuellen Datenbankzustand beschreiben, beginnen bei ORACLE mit dem Präfix V\$.

5.9 Datenkontrolle

Die Datenkontrollmechanismen in einem relationalen Datenbanksystem umfassen Maßnahmen der Zugriffskontrolle wie die Vergabe von Zugriffsrechten und die Bereitstellung bestimmter, meist eingeschränkter, Sichten auf die Daten. Zur Datenkontrolle gehört aber auch das Festlegen von Integritätsbedingungen, die bestimmen, welche Daten in der Datenbank zulässig sind.

Das Datenbanksystem hat die Aufgabe, die so festgelegten Anforderungen an die Datenhaltung bei jedem Befehl zu überprüfen, wobei für Transaktionen insbesondere das ACID-Prinzip zu erfüllen ist.¹²

5.9.1 Sichten

Benutzersichten oder Views sind, wie es der Begriff ausdrückt, individuelle Blickwinkel von Benutzern auf die Daten von Tabellen. Eine View ist dabei eine Art virtuelle Tabelle, sie beinhaltet keine eigenen Datensätze, kann aber innerhalb der DML genauso wie eine Tabelle verwendet werden. Gründe für Views sind:

- ▶ Für jeden Benutzer einer Datenbank können die Informationen so zur Verfügung gestellt werden, wie er sie individuell für seine Tätigkeit benötigt, unabhängig von der physischen Struktur der Datentabellen.
- ▶ Komplizierte Anfragen, in denen Tabellen verknüpft oder vereinigt werden oder die aus Unterabfragen aufgebaut sind, können in der View vordefiniert werden, so dass der Anwender nur die Grundselektionsmöglichkeiten kennen muss.
- ▶ Benutzersichten sind im Zusammenhang mit Benutzerrechten ein wirksames Werkzeug zum Aufbau eines Datenschutzsystems. Wenn eine Tabelle Informationen enthält, die nicht jedem zugänglich sein dürfen, kann eine View in der Weise erstellt werden, dass sie nur die unbedenklichen Informationen enthält.
- ▶ Views gewährleisten eine erhöhte Datenunabhängigkeit. Selbst wenn der Aufbau einer Tabelle völlig verändert wird (beispielsweise durch nachträgliches Normalisieren) können Programme ohne Änderung oder Neuübersetzung weiterarbeiten, wenn Views die ursprüngliche Sicht der Daten wiederherstellen.

Eine View beinhaltet physisch keinerlei Daten. Sie verkörpert nur eine Vorschrift, wie die View-einträge aus Basistabellen gewonnen werden. Diese Vorschrift entspricht dabei einem SELECT:

```
/—————\  
CREATE VIEW Viewname [(Spaltenliste)]  
    AS SELECT-Anweisung  
    [WITH CHECK OPTION]  
—————/
```

Die optionale Spaltenliste benennt die Spalten, die in der virtuellen Tabelle erscheinen sollen. Dabei können die Spalten der Views andere Namen erhalten, als ursprünglich für die Basistabellen definiert wurden. Es können auch neue Spalten aufgenommen werden, die in den Basistabellen nicht existieren, sondern durch arithmetische Ausdrücke und Funktionen ermittelt werden. Fehlt die Spaltenliste, werden die in der SELECT-Anweisung genannten Attribute der Quelltabelle mit ihren Originalnamen in die View übernommen.

¹²Die Befehle COMMIT und ROLLBACK sind deshalb eigentlich SQL-Befehle, die in den Bereich der Datenkontrolle gehören.

Innerhalb der SELECT-Anweisung können alle zuvor vorgestellten Möglichkeiten eingesetzt werden. Ausnahme ist hier ORDER BY, diese Klausel wird häufig nicht unterstützt, da Relationen keine Reihenfolge haben und eine Sortierung bei der Anfrage auf die View angegeben werden kann. Die Option 'WITH CHECK OPTION' bewirkt, dass eine View, die auch für INSERT- und UPDATE-Operationen zugelassen ist, dies nur zulässt, wenn sich die neuen Daten innerhalb des Viewausschnitts befinden. Das heißt, dass der neue Datensatz nur dann in die Basistabelle eingespeichert wird, wenn er die WHERE-Klausel der View erfüllt.

Beispiel:

```
⇒ CREATE VIEW ArmeProgrammierer
  (PNR, Name, Beruf, Gehalt, ANR) AS
    SELECT PNR, Name, Beruf, Gehalt, ANR FROM Pers
    WHERE Beruf='Programmierer' AND Gehalt < 40000
```

Änderungen an einer View mit INSERT und UPDATE können nur unter folgenden Bedingungen durchgeführt werden:

- ▶ Die View darf sich nur über eine Tabelle erstrecken.
- ▶ DISTINCT, GROUP BY und HAVING dürfen in der Viewdefinition nicht vorkommen.
- ▶ In der Spaltenliste der SELECT-Anweisung dürfen nur Spalten genannt werden, die ihren Ursprung in der Basistabelle haben.
- ▶ Die SELECT-Anweisung darf nicht DISTINCT enthalten.

Bei einer Selektion auf eine VIEW wird das SELECT mit dem SELECT der View verknüpft, danach wird die Gesamtselektion auf den Basistabellen ausgeführt.

Der ALTER-Befehl für eine Sicht erlaubt nur eine Neuübersetzung der View:

```
/—————\  
  ALTER VIEW View COMPILE  
—————/
```

Sichten eignen sich auch sehr gut zur Modellierung von Generalisierungen. Generalisierungen sind durch zwei Eigenschaften gekennzeichnet: Inklusion und Vererbung. Objekte eines Untertyps einer Generalisierungshierarchie sollen auch automatisch zu ihrem Obertyp gehören und die Attribute des Obertyps erben. Eine Lösung dieses Problems bietet die Überrelation (vgl. Abschnitt 3.4.3.4). Darauf aufbauend kann man die einzelnen Untertypen als Sichten der Überrelation darstellen:

Beispiel:

```
⇒ CREATE VIEW Java-Entwickler AS
  SELECT PNR, NAME, LetzteSchulung, E-Umgebung
  FROM Mitarbeiter WHERE Typ = 'J'
```

Mit Hilfe dieser Views wird die Darstellung der Hierarchie übersichtlich und alle Vorteile der Überrelation bleiben erhalten. Diese Form der Generalisierung wird deshalb in den meisten Fällen verwendet. Aber auch beim Hausklassenmodell können mit Hilfe von Views die Anfragenachteile beseitigt werden. Wird der Name zu einer beliebigen PNR gesucht, so mussten alle Hierarchierelationen durchsucht werden. Eine View kann diesen Vorgang zusammenfassen.

Beispiel:

```
⇒ CREATE VIEW Mitarbeiter (PNR, NAME) AS
  (SELECT PNR, NAME FROM Manager) UNION
  (SELECT PNR, NAME FROM Entwickler) UNION
  (SELECT PNR, NAME FROM Java-Entwickler) UNION
  (SELECT PNR, NAME FROM C++-Entwickler)
```

5.9.2 Zugriffsrechte

SQL unterscheidet zwei Arten von Zugriffsrechten. So gibt es allgemeine Zugriffsrechte, die Systemprivilegien, die es einem Benutzer beispielsweise erlauben, eine Tabelle anzulegen. Daneben gibt es noch Objektprivilegien, die den Zugriff auf einzelne Datenbankobjekte, wie Tabellen, regeln. Zur Erleichterung der Rechtevergabe bei einer großen Benutzeranzahl können Regeln zu Rollen zusammengefasst werden. Ein Beispiel ist die DBA-Rolle bei ORACLE, die alle Rechte zur Datenbankadministration zusammenfasst. Die Vergabe einer Rolle zu einem Benutzer entspricht syntaktisch der Vergabe eines Systemprivilegs.

5.9.2.1 Vergabe eines Systemprivilegs

```
/-----\
GRANT {Privileg | Rolle} TO
{Benutzername | Rolle | PUBLIC}
[WITH GRANT OPTION]
\-----/
```

Ist die Option WITH GRANT OPTION angegeben, kann ein Benutzer die so erhaltenen Rechte an andere Benutzer weitergeben. Durch das Zuweisen von Privilegien an eine Rolle kann die Rolle entsprechend definiert werden. Rechte können aber auch PUBLIC vergeben werden, so dass dieses Recht jedem Benutzer zukommt. Die Systemprivilegien sind systemabhängig, beispiele für Systemprivilegien bei ORACLE sind:

ALTER ANY TABLE: Erlaubt jede Tabelle oder Sicht zu ändern.

ALTER DATABASE: Erlaubt die Datenbankkonfiguration zu ändern

CREATE TABLE: Erlaubt Tabellen im eigenen Schema zu erzeugen. Um eine Tabelle zu erzeugen, müssen die Berechtigten außerdem über Platzquoten auf dem Tabellenbereich für die Aufnahme der Tabelle verfügen.

CREATE ANY TABLE: Erlaubt Tabellen in einem beliebigen Schema zu erzeugen. Der Eigentümer des Schemas, das die Tabelle enthält, muss über Platzquoten auf dem Tabellenbereich zur Aufnahme der Tabelle verfügen.

EXECUTE ANY PROCEDURE: Erlaubt Prozeduren oder Funktionen aufzurufen.

SELECT ANY TABLE: Erlaubt Tabellen, Sichten oder Snapshots in einem beliebigen Schema abzufragen.

5.9.2.2 Entzug eines Systemprivilegs

```
/-----\
REVOKE {Privileg | Rolle}
      FROM {Benutzername | Rolle | PUBLIC}
\-----/
```

5.9.2.3 Vergabe eines Objektprivilegs

```
/-----\
GRANT {Objektprivileg | ALL [PRIVILEGES] | Rolle}
      [ (Spalte [, Spalte]...) ]
      ON [Schema.]Objekt
\-----/
```

```

TO {Benutzer | Rolle | PUBLIC}
[WITH GRANT OPTION]
\-----/
|-----|
REVOKE [GRANT OPTION FOR] Objektprivileg
ON [Schema.]Objekt
FROM {Benutzer | Rolle | PUBLIC}
\-----/

```

Das Privileg bezeichnet jeweils den Typ von SQL-Operationen, die der Benutzer ausführen darf. Mit ALL werden alle Privilegien für ein Objekt auf einmal erteilt. ORACLE unterstützt beispielsweise folgende Objektprivilegien:

- ▶ ALTER
- ▶ DELETE
- ▶ EXECUTE
- ▶ INDEX
- ▶ INSERT
- ▶ READ
- ▶ REFERENCES
- ▶ SELECT
- ▶ UPDATE
- ▶ ALL

Attributeinschränkung sind nur bei SELECT, INSERT, UPDATE und REFERENCES möglich.

Beispiele:

⇒ GRANT SELECT ON Pers TO Max

Das Max zugeteilte Recht SELECT erlaubt ihm, alle Daten aus der Tabelle Pers zu selektieren.

⇒ GRANT SELECT, UPDATE(Gehalt) ON Pers TO Heike

Heike wird das Lesen der Tabelle Pers sowie das Ändern der Gehaltseinträge gestattet.

Beim Rechteentzug bewirkt der Zusatz GRANT OPTION FOR, dass nur das Recht auf Weitergabe entzogen wird, das Recht an sich bleibt beim Benutzer weiterhin bestehen. Der optionale Zusatz RESTRICT bedeutet, dass die Revoke-Anweisung nicht durchgeführt wird, wenn der betreffende Benutzer seine Privilegien an andere weitergegeben hat. Durch die Angabe von CASCADE werden auch alle weitergegebenen Privilegien zurückgenommen.

5.9.2.4 Zugriffsrechte und Views

Das Leserecht auf eine Tabelle kann nicht direkt auf bestimmte Einträge der Tabellen eingeschränkt werden. Ist dies erforderlich, muss man die Vergabe der Leserechte über Views realisieren. Dazu wird eine View definiert, die genau die Spalten und Zeilen einer Tabelle beschreibt, die einem Benutzer oder einer Benutzergruppe zugänglich sein soll. Der lesende Zugriff auf die Originaltabelle wird diesen Benutzern verboten, der Zugriff auf die View jedoch gestattet. Dadurch erhalten die Benutzer nur die Informationen, die für sie vorgesehen sind.

Beispiel:

```
⇒ CREATE VIEW PersAllgemeinAbt1 AS
    SELECT PNR, Name, Wort FROM Pers
    WHERE ANR = 'A1'
    GRANT SELECT ON PersAllgemeinAbt1 TO Max
```

Die View PersAllgemeinAbt1 enthält nur unkritische Personaldaten, so fehlt beispielsweise das Gehalt. Weiterhin ist die View auf die Abteilung A1 beschränkt, so dass der Benutzer Max nur diese Informationen lesen darf.

5.9.2.5 Rollen

Die Verwaltung von Benutzerrechten wird bei einer großen Benutzeranzahl schnell aufwändig. Haben verschiedene Benutzer aufgrund ihrer Tätigkeit dieselben Rechte, können diese Privilegien zu einer Rolle zusammengefasst werden. Die Rechte werden dabei einer Rolle zugewiesen und diese Rollen können dann wieder an Benutzer verteilt werden. Ein Benutzer hat dann alle Privilegien der Rollen, die ihm zugeordnet sind.

Syntaktisch kann hierfür beim Grant-Befehl der Empfänger eines Privileges eine Rolle sein und neben den Standardrechten können Rollen Benutzer mit Grant zugeordnet werden. Erzeugt wird eine Rolle über den Befehl:

```
/—————\  
CREATE ROLE rolle  
—————/
```

5.9.3 Integritätskontrolle

Eine Transaktion ist eine Folge von Datenbankoperationen, welche die Datenbank von einem logisch konsistenten in einen neuen logisch konsistenten Zustand überführt. Die Aufgaben der Integritätskontrolle ist die Überwachung und Einhaltung der logischen Datenbankkonsistenz. Grundsätzlich lassen sich diese Aufgaben entweder auf Anwendungsebene oder zentral durch das Datenbanksystem realisieren. Mit der ACID-Eigenschaft ist das Datenbanksystem für die Integritätskontrolle verantwortlich. Diese zentrale Definition und Überwachung von Integritätsbedingungen ermöglicht eine Vereinfachung der Anwendungsentwicklung, da die entsprechenden Prüfungen nicht redundant in zahlreichen Programmen zu integrieren ist. Die Wartbarkeit derartiger Lösungen ist zusätzlich mit einem hohen Aufwand verbunden. Auch kann die DBS-interne Überprüfung von Integritätsbedingungen zu Leistungsvorteilen führen, da Aufrufe an das Datenbanksystem eingespart werden. Schließlich wird auch eine Integritätskontrolle für DB-Änderungen erreicht, die direkt (ad hoc) vorgenommen werden. In kommerziellen Systemen stehen deshalb Mechanismen zur Integritätskontrolle bereit, zumal im SQL-92-Standard umfassende Möglichkeiten zur Spezifikation von Integritätsbedingungen festgelegt wurden.

5.9.3.1 Arten von Integritätsbedingungen

Semantische Integritätsbedingungen lassen sich hinsichtlich mehrerer Kategorien klassifizieren:

Modellinhärente vs. sonstige Integritätsbedingungen

Modellinhärente Bedingungen folgen aus der Strukturbeschreibung des jeweiligen Datenmodells und sind somit für alle Anwendungen zu gewährleisten. Im Falle des relationalen Datenmodells sind dies die relationalen Invarianten, also die Primärschlüsseleigenschaft sowie die referentielle Integrität für Fremdschlüssel. Zudem sind die zulässigen Werte eines Attributs durch einen Definitionsbereich zu beschränken.

Reichweite

Wichtige Fälle mit jeweils unterschiedlicher Datengranularität sind:

- ▶ Attribut (Geburtsjahr > 1900)
- ▶ Satzausprägung (Geburtsdatum < Einstellungsdatum)
- ▶ Satztyp (Eindeutigkeit von Attributwerten)
- ▶ Satztypübergreifend (Fremdschlüssel über Tabellen)

Zeitpunkt der Überprüfbarkeit

Bestimmte Integritätsbedingungen, wie etwa Wertebereichsbeschränkungen von Attributen, können unmittelbar bei der Ausführung von Änderungsoperationen vom DBS geprüft werden. Zur Einhaltung der ACID-Eigenschaft reicht es aber aus, wenn am Transaktionsende gewährleistet ist, dass alle Integritätsbedingungen erfüllt sind. Einzelne Bedingungen kann man daher auch erst am Transaktionsende überprüft lassen (verzögerte Integritätsbedingungen).

statische vs. dynamische Bedingungen

Statische Bedingungen (Zustandsbedingungen) beschränken zulässige Zustände der Datenbank (beispielsweise Gehalt < 500000), während dynamische Integritätsbedingungen (Übergangsbedingungen) zulässige Zustandsübergänge festlegen. Beispiele hierfür sind:

- ▶ Familienstand darf nicht direkt von ledig in geschieden geändert werden.
- ▶ Gehalt darf nicht kleiner werden.
- ▶ Gehalt darf in drei Jahren nicht mehr als 25 Prozent steigen.

5.9.3.2 Integritätsbedingungen in Tabellendefinitionen

Viele Integritätsbedingungen, die nur die Daten einer Tabelle betreffen, können direkt beim `CREATE TABLE` dieser Tabelle angegeben werden. Dies beinhaltet:

- ▶ Definition eines Schlüsselkandidaten
- ▶ Definition eines Fremdschlüssels
- ▶ Definition einer Prüfbedingung

Jeder dieser Bedingung kann optional das Schlüsselwort `CONSTRAINT` gefolgt von dem Namen der Bedingung vorangestellt werden. Für den Fall, dass die Bedingung für ein Attribut gilt, wurde der Einsatz dieser Möglichkeiten schon beim `CREATE TABLE` vorgestellt. Es können aber auch Integritätsbedingungen formuliert werden, die mehrere Spalten derselben Tabelle betreffen. Hierzu ist die Integritätsbedingung in einer eigenen Zeile innerhalb der Tabellendefinition einzutragen. So kann beispielsweise auch die Primärschlüssel- und `UNIQUE`-Eigenschaft für mehrere Attribute angegeben werden.

CHECK erlaubt die Formulierung von Bedingungen für die Datensätze, indem ein zu erfüllender Bedingungsausdruck angegeben wird. In SQL92 kann diese Bedingung eine beliebige WHERE-Bedingung sein und damit auch Unterabfragen beispielsweise über EXISTS oder IN enthalten. Dies ist notwendig, wenn die Integritätsbedingungen tabellenübergreifend sind. Viele Datenbanksysteme unterstützen diese Möglichkeit jedoch nicht.

5.9.3.3 Allgemeine Bedingungen

Allgemeine Integritätsbedingungen, die nicht nur eine Tabelle betreffen können mit Assertions realisiert werden.

```
/—————\  
CREATE Assertion Aname  
    CHECK (Bedingung)  
    [ DEFERRED — IMMEDIATE ]  
—————/
```

Die Bedingung kann dabei in Form einer WHERE-Bedingung formuliert werden. Das Datenbanksystem gewährleistet, dass die Bedingung immer erfüllt ist. Die meisten Bedingungen beginnen bei Assertions mit `not exists`, da dadurch Eigenschaften formulierbar sind, dass alle Datensätze zu erfüllen haben.

Die optionale Angabe von `deferred` bzw. `immediate` bestimmt den Überprüfungszeitpunkt, `deferred` überprüft die Integritätsbedingung am Transaktionsende, `immediate` nach jedem Befehl.

Beispiele:

```
⇒ CREATE ASSERTION MinGehalt CHECK  
    ((SELECT MIN(Gehalt) FROM Pers) > 30000)  
Das Mindestgehalt liegt bei 30000 Euro.
```

```
⇒ CREATE ASSERTION PosBudget CHECK  
    NOT EXISTS (SELECT * FROM ABT  
    WHERE BUDGET < (SELECT SUM (GEHALT)  
        FROM Pers WHERE Pers.ANR = Abt.ANR))  
DEFERRED
```

Das Budget einer Abteilung ist mindestens so groß wie die Gehaltssumme der Abteilungsmitarbeiter. Dies ist eine tabellenübergreifende Bedingung, die innerhalb einer CREATE TABLE-Anweisung meist nicht direkt angegeben werden kann.

Die Angabe von `deferred` erlaubt, dass ein neuer Mitarbeiter eingetragen wird, bevor das Budget erhöht wird.

Assertions können mit `DROP ASSERTION AName` gelöscht werden.

5.9.3.4 Überprüfungszeitpunkte für Integritätsbedingungen

Das Transaktionsparadigma verlangt von einem Datenbanksystem, dass am Ende einer Transaktion alle angegebenen Integritätsbedingungen erfüllt sind. Dabei wird nicht festgelegt, wann der Test der Bedingungen durchgeführt wird. Dies kann nach jedem Befehl oder erst am Transaktionsende erfolgen. Gibt es beispielweise zyklische Abhängigkeiten mehrerer Tabellen, so dass eine Änderung an einer Tabelle auch eine Änderung an anderen Tabellen zu Folge hat, können die Integritätsbedingungen nur am Ende erfüllt sein. Innerhalb der Transaktion ist kurzzeitig ein inkonsistenter Datenbankzustand vorhanden.

Die Definition von Assertions erlaubt die Angabe des Überprüfungszeitpunkts. Andere Integritätsbedingungen werden normalerweise sofort nach jeder SQL-Anweisung geprüft. SQL92 enthält den Befehl `SET CONSTRAINTS`, mit dem auch diese Integritätsbedingungen verzögert überprüfbar werden.

```
/-----\  
SET CONSTRAINTS { Kommaliste | ALL }  
    { DEFERRED | IMMEDIATE }  
-----\ /
```

Der Ausführungszeitpunkt der Prüfung wird hierdurch nur für die aktuelle Transaktion gesetzt. Man gibt wahlweise die Namen der Bedingungen (Constraints) an. Dies erfordert auch, dass man den Bedingungen mit der CONSTRAINT-Option Namen gegeben hat.

5.9.3.5 Trigger

Das Konzept der Integritätsregeldefinition mit Assertions konnte sich bisher nicht durchsetzen. Heutige Datenbanksysteme unterstützen jedoch eine allgemeinere Version derartiger Regeln, die auch für andere Aufgaben in der Datenbankverwaltung eingesetzt werden können, die so genannten Trigger (Auslöser). Vereinfacht formuliert wurde die Idee der integritätssichernden Regeln von der automatischen Prüfung zu transaktionsbezogenen Zeitpunkten getrennt. Das Ergebnis waren Regeln, deren Überprüfung bei bestimmten Datenbankaktionen gefeuert werden und für verschiedene Zwecke eingesetzt werden können.

Ein Trigger führt automatisiert definierbare Aktionen aus, wenn ein Ereignis eintritt. Trigger sind damit ein sehr mächtiges Werkzeug zur Sicherung der Datenintegrität, sie übersteigen die Möglichkeiten der einfachen Integritätsbedingungen. Man kann sie zusätzlich dafür verwenden, Folgeaktionen für eine Ausgangsaktion zu definieren um beispielsweise Aktionen zu protokollieren oder Informationen nach außen weiterzugeben.

Zur Erzeugung eines Triggers sind folgende Teile zu spezifizieren:

Name: Wie eine Tabelle hat auch ein Trigger einen eindeutigen Namen innerhalb eines Schemas.

Auslösende Ereignisse: Das auslösende Ereignis ist das Ereignis, durch das der Trigger aktiviert (gefeuert) wird. Im Allgemeinen ist das auslösende Ereignis ein Einfügen, Löschen oder Aktualisieren von Zeilen in einer spezifischen Tabelle. Ist der Auslöser ein Update, kann sich dieser auf sämtliche Spalten der Tabelle oder nur auf einzelne davon beziehen. Ein Trigger ist der im auslösenden Ereignis genannten Tabelle zugeordnet. Der Trigger wird auch dann ausgelöst, wenn die Tabelle über eine auf ihr definierte Sicht manipuliert wird.

Aktivierungszeitpunkt: Der Zeitpunkt der Aktivierung eines Triggers liegt bevor oder nachdem der Befehl, der das Ereignis auslöst, vom Datenbanksystem durchgeführt wurde. Ein Trigger vor einer INSERT, UPDATE oder DELETE-Anweisung wird meist für Integritätstests oder zur maschinellen Erzeugung von Spaltenwerten eingesetzt. After-Trigger dienen zur Propagierung von Änderungen, zum Versenden von Nachrichten oder zum Protokollieren der Operation. Ein After-Trigger wird stets erst dann ausgeführt, wenn die auslösende SQL-Anweisung ausgeführt und alle ihre Integritätsbedingungen erfolgreich getestet werden konnten. Der Trigger sieht somit nicht nur die Effekte der auslösenden Anweisung, sondern auch die von kaskadierenden Änderungen und Löschungen von Fremdschlüsselbedingungen. Entsprechend kann ein After-Trigger einen eingefügten oder geänderten Datensatz auch nicht mehr verändern. Wird von einer Anweisung mehr als ein After-Trigger aktiviert, so sieht jeder dieser die von den Triggern, die vor ihm aktiviert wurden, auf der Datenbank verursachten Effekte.

Ein besonderer Zeitpunkt ist 'INSTEAD OF'. Hier wird der SQL-Befehl nicht ausgeführt, stattdessen nur das Triggerprogramm. Hiermit ist es beispielsweise auch möglich, Trigger auf nicht änderbare Views zu definieren um damit Änderungsoperatoren auf Views zuzulassen, die dann durch den Trigger passende Operationen in der Datenbasis umgesetzt werden. Auch kann ein Instead-of-Trigger auf eine Delete-Ereignis erreichen, dass der Datensatz nicht gelöscht wird, sondern nur als gelöscht markiert wird.

Granularität: Die SQL-Anweisung, die das triggernde Ereignis verursacht hat, kann eine oder mehrere Zeilen in der Tabelle betreffen (UPDATE, DELETE). Der den Trigger definierende Benutzer kann angeben, ob der Trigger in einem solchen Fall einmal für die gesamte Anweisung (Statement-Trigger) oder einmal für jede Zeile, die verändert wird (Row-Trigger), aktiviert werden soll. Bei IBM DB2 müssen Before-Trigger stets Zeilentrigger sein.

Transaktionsvariablen: Wenn ein Trigger aktiviert wird, muss er häufig von Informationen über die spezifische Datenbankänderung Gebrauch machen, die ihn aktiviert hat. So muss beispielsweise ein UPDATE-Row-Trigger die Datenwerte in den aktualisierten Zeilen vor und nach dem UPDATE sehen. Analog muss ein DELETE-Trigger alle zu löschen Zeilen sehen. Diese Art transitionaler Informationen wird einem Trigger über Transaktionsvariablen zugänglich gemacht, von denen es vier Arten gibt:

- ▶ Die *Old-Row-Variable* repräsentiert den Wert der modifizierten Zeile vor dem auslösenden Ereignis. Auf die Zeilenelemente kann man über `.' zugreifen.
- ▶ Die *New-Row-Variable* repräsentiert den Wert der modifizierten Zeile nach dem auslösenden Ereignis, also mit veränderten Werten.
- ▶ Die *Old-Table-Variable* repräsentiert den hypothetische Read-Only-Tabelle, die sämtliche modifizierten Zeilen vor dem auslösenden Ereignis enthält.
- ▶ Die *New-Table-Variable* repräsentiert den hypothetische Read-Only-Tabelle, die sämtliche modifizierten Zeilen nach dem auslösenden Ereignis enthält.

Wenn man einen Trigger definiert, kann man Transaktionsvariablen in einer optionalen REFERENCING-Klausel für das Triggerprogramm zugänglich definieren und dabei diesen Variablen einen beliebigen Namen geben. Eine Triggerdefinition kann mehr als eine Transaktionsvariable enthalten, jedoch höchstens eine von jedem Typ.

Nicht immer ist es sinnvoll alle Arten von Transaktionsvariablen in ein Programm aufzunehmen. Ein Insert-Trigger kann nur NEW-Variablen benutzen, ein DELETE-Trigger nur OLD-Variablen. Dies ist in Tabelle 5.3 zusammengefasst.

Triggerbedingung: Eine Triggerbedingung ist ein Test, der zu wahr, falsch oder unbekannt ausgewertet werden kann. Er kann ähnlich wie eine WHERE-Klausel ein oder mehrere Prädikate enthalten. Eine Triggerbedingung kann Transaktionsvariablen und Unteranfragen enthalten. Wenn der Trigger aktiviert wird, wird der Triggerrumpf nur ausgeführt, wenn die Triggerbedingung zu wahr ausgewertet wird.

Triggerrumpf: Der Triggerrumpf besteht aus Anweisungen, er definiert das Programm das ausgeführt werden soll, wenn das Ereignis bei erfüllter Bedingung eintritt. Falls eine der im Triggerrumpf enthaltenen SQL-Anweisungen scheitert, werden das SQL-Statement, das das Ereignis ausgelöst hat, sowie alle Aktionen sämtlicher dadurch ausgelöster Trigger zurückgesetzt. Obwohl das auslösende Statement zurückgesetzt wird, wird die aktuelle Transaktion als noch laufend betrachtet, es können weitere Anweisungen sowie ein abschließendes COMMIT oder ROLLBACK folgen.

5.9.3.5.1 Erzeugen und Löschen von Triggern

```
/-----\
CREATE TRIGGER Triggername
{[NO CASCADE]13 BEFORE | AFTER | INSTEAD OF}
{INSERT | DELETE | UPDATE [OF Spalte [,Spalte]...]}
ON Tabellename
```

¹³Der Zusatz NO CASCADE gibt es nur bei IBM DB2 und muss bei BEFORE-Triggern angegeben werden. Dies dient als Erinnerung dafür, dass eine BEFORE-Trigger niemals einen anderen BEFORE-Trigger aktiviert.

Ereignis	Zeiletrigger	Anweisungstrigger
BEFORE INSERT	New Row	ungültig
BEFORE UPDATE	Old Row, New Row	ungültig
BEFORE DELETE	Old Row	ungültig
AFTER INSERT	New Row, New Table	New Table
AFTER UPDATE	Old Row, New Row Old Table, New Table	Old Table New Table
AFTER DELETE	Old Row, Old Table	Old Table

Tabelle 5.3: Mögliche Transitionsvariablen bei Triggerarten

```
[REFERENCING Transvar [,Transvar]...]
{FOR EACH STATEMENT | FOR EACH ROW}
[MODE DB2SQL]14
[WHEN Triggerbedingung]
Triggerrumpf
```

\—————/

Transaktionsvariablen werden wie folgt deklariert:

/—————\ {OLD | NEW | OLD_TABLE | NEW_TABLE } [AS] Varname
 \—————/

Wird ein Trigger nicht länger benötigt, kann er über die DROP-Anweisung gelöscht werden.

/—————\ DROP TRIGGER triggername
 \—————/

5.9.3.5.2 Die Signal-Anweisung Eine SIGNAL-Anweisung hat den Zweck, eine Fehlerbedingung auszulösen und die durchgeführten Änderungen einer SQL-Anweisung zurückzusetzen. Dies kann sowohl in BEFORE- als auch AFTER-Triggern verwendet werden. Die Ausgangstransaktion wird davon nicht beeinflusst, sondern nur der Teil, der zur Auslösung des Triggers führte. Dadurch kann der Benutzer wählen, ob er die anderen Anweisungen bestätigt (COMMIT) oder auch zurücksetzen möchte (ROLLBACK).

/—————\ SIGNAL SQLSTATE zustand (nachricht)
 \—————/

Der in einer SIGNAL-Anweisung spezifizierte Zustand muss bei IBM DB2 als Zeichenkettenliteral mit genau fünf Zeichen (beispielsweise '70ABC') ausgedrückt werden. Die fünf Zeichen müssen Ziffern oder Großbuchstaben sein. Bei der Auswahl eines SQLSTATE zur Darstellung einer benutzerdefinierten Fehlerbedingung sollte man Werte vermeiden, die im SQL92-Standard oder vom Datenbankhersteller reserviert sind. Man kann Konflikte leicht dadurch vermeiden, dass man einen SQLSTATE mit einer Ziffer zwischen 7 und 9 oder einen Buchstaben zwischen I und Z beginnen lässt. Dies ist aber keine Vorschrift. Die in einer SIGNAL-Anweisung spezifizierte Nachricht kann ein Ausdruck mit einem maximal 70-Zeichen langen Ergebnis sein.

¹⁴ MODE DB2SQL muss bei IBM DB2 angegeben werden. Dies repräsentiert den Triggerausführungsmodus, der in DB2 implementiert ist. Dieser Zusatz stellt sicher, dass existierende Applikationen nicht von alternativen Triggerausführungsmodi, die es möglicherweise in Zukunft geben wird, betroffen sein wird.

Wenn eine SIGNAL-Anweisung innerhalb eines Triggerrumpfes ausgeführt wird, wird die den Trigger auslösende SQL-Anweisung sowie jede aus dieser resultierende Änderung zurückgesetzt. Die Anwendung, die die triggernde Anweisung ausgeführt hat, erhält den in der SIGNAL-Anweisung spezifizierten SQLSTATE und die Nachricht (sowie einen SQLCODE -438 bei IBM DB2).

Wenn man einen Trigger schreibt, der eine Funktion aufruft, die eine externe Aktion wie das Versenden einer Nachricht ausführt, und eine Anweisung, die den Trigger auslöst, zunächst ausgeführt und dann zurückgesetzt wird, hat das Datenbanksystem keine Möglichkeit, auch die externe Aktion zurückzusetzen. Daher muss man mit der Benutzung solcher Funktionen ausgesprochen vorsichtig sein und einen Mechanismus zur Erzeugung kompensierender externer Aktionen im Falle eines Rollbacks vorsehen.

5.9.3.5.3 Integritätsbedingungen versus Trigger

Grundsätzlich ist die Verwendung von Integritätsbedingungen einem Trigger vorzuziehen. Dies hat folgende Gründe:

- ▶ Integritätsbedingungen sind weniger prozedural als Trigger und geben dem System mehr Möglichkeiten zur Optimierung.
- ▶ Integritätsbedingungen werden im Unterschied zu Triggern zum Zeitpunkt ihrer Erzeugung für alle in der Datenbank existierenden Daten durchgesetzt.
- ▶ Integritätsbedingungen schützen Daten vor jeder Art Anweisung gegen ein Versetzen in einen ungültigen Zustand, wohingegen Trigger nur für bestimmte Anweisungen wie UPDATE oder DELETE gelten.

Auf der anderen Seite sind Trigger mächtiger als Integritätsbedingungen und können Regeln durchsetzen, bei denen Integritätsbedingungen überfordert sind. So sind für Zustandsübergangsbedingungen Trigger die einzige Möglichkeit.

5.9.3.5.4 Interaktion zwischen Integritätsbedingungen und Triggern

Wenn man Integritätsbedingungen und Trigger verwendet, sollte man sich darüber im klaren sein, in welcher Reihenfolge diese bei der Verarbeitung einer SQL-Anweisung angewendet bzw. ausgeführt werden. Eine einzelne INSERT, DELETE oder UPDATE-Anweisung kann viele Datenzeilen verändern. Auf jede dieser Zeilen kann eine Liste von Integritätsbedingungen und Triggern anwendbar sein. Die Abfolge der Ereignisse ist bei IBM DB2 wie folgt, andere Datenbanksysteme arbeiten sehr ähnlich:

1. Die SQL-Anweisung wird 'hypothetisch' ausgeführt. Dabei wird eine Änderungsliste mit den alten sowie den neuen Werten aller Zeilen, die von der Anweisung verändert würden, erstellt. Die auf dieser Liste vermerkten Änderungen werden noch nicht auf die Datenbank angewendet.
2. Die von der Anwendung aktivierten BEFORE-Trigger werden in der Reihenfolge ihrer Erzeugung ausgeführt. Before-Trigger reagieren dabei nicht direkt auf der Datenbank, aber sie können (neue) Werte auf der Änderungsliste verändern.
3. Die Änderungen der Änderungsliste werden in der Datenbank durchgeführt.
4. Integritätsbedingungen werden überprüft und alle Datenbankänderungen für die betreffende Anweisung werden zurückgesetzt, falls eine Verletzung entdeckt wird. Fremdschlüsselbedingungen mit Löschaktion wie CASCADE oder SET NULL können weitere Datenbankänderungen verursachen. Jede dieser sekundären Änderungen wird wie folgt ausgeführt:
 - ▶ Die Sekundäränderung wird hypothetisch ausgeführt.

- ▶ Von der Sekundäränderung aktivierte BEFORE-Trigger werden in der Reihenfolge ihrer Erzeugung ausgeführt, wodurch möglicherweise neue Werte auf der Änderungsliste modifiziert werden.
 - ▶ Die Änderungsliste wird in der Datenbank abgearbeitet und außerdem mit der Änderungsliste der ursprünglichen SQL-Anweisung gemischt.
 - ▶ Integritätsbedingungen werden überprüft und alle Datenbankänderungen werden zurückgesetzt, falls eine verletzt wird. Diese Abfolge von Ereignissen wird solange wiederholt, bis es keine Sekundäränderungen mehr gibt. An diesem Punkt befindet sich die Datenbank in einem konsistenten Zustand und sämtliche Änderungen sind in einer Hauptänderungsliste zusammengetragen.
5. Jetzt werden After-Trigger, die von der ursprünglichen SQL-Anweisung oder einer der sekundären Änderungen ausgelöst wurden, ausgeführt, wieder in der Reihenfolge ihrer Erzeugung. Jeder After-Statement-Trigger wird genau einmal ausgeführt, selbst wenn keine Zeilen verändert wurden. Jeder After-Row-Trigger wird genau einmal für jede Zeile, die entweder von der ursprünglichen SQL-Anweisung oder einer sekundären Änderung verändert wurde, ausgeführt. Die Old- sowie New-Variablen eines jeden Triggers sind über die Werte auf der Änderungsliste definiert.
- Ein After-Trigger kann seinerseits SQL-Anweisungen enthalten und die Datenbank verändern. Jeder solche Trigger kann die von anderen Triggern, die vorher ausgeführt wurden, in der Datenbank verursachten Effekte sehen.
- Da der Rumpf eines After-Triggers aus einer Folge von SQL-Anweisungen besteht, kann jede dieser ihrerseits Integritätsbedingungen und Trigger feuern. Jede SQL-Anweisung innerhalb eines Triggers wird unabhängig ausgeführt, unter Verwendung der gleichen obigen Reihenfolge der Ereignisse. Als Konsequenz daraus befindet sich die Datenbank nach Ausführung jeder einzelnen, in einem Triggerrumpf vorkommenden SQL-Anweisung definitiv in einem konsistenten Zustand.

5.9.3.5.5 Beispiele mit IBM DB2 UDB

```

1 CREATE TRIGGER ang_trig
2   NO CASCADE BEFORE INSERT ON Pers
3   REFERENCING NEW AS neuezeile
4   FOR EACH ROW MODE DB2SQL
5   SET (gehalt, bonus) =
6     (SELECT gehalt, bonus FROM Anfangsgehaelter
7      WHERE jobcode = neuezeile.jobcode);

```

Listing 5.1: Setzen des Einstellungsgehalts

```

1 CREATE TRIGGER ang_trig2
2   NO CASCADE BEFORE UPDATE OF Gehalt ON Pers
3   REFERENCING NEW AS neuezeile
4       OLD AS altezeile
5   FOR EACH ROW MODE DB2SQL
6   WHEN (neuezeile.gehalt < 1.05 * altezeile.gehalt)
7   SET neuezeile.gehalt = 1.05 * altezeile.gehalt;
8

```

Listing 5.2: Mindestgehaltserhöhung von 5%

```

1  CREATE TRIGGER ang_trig3
2    NO CASCADE BEFORE DELETE ON Pers
3    REFERENCING OLD AS altezeile
4    FOR EACH ROW MODE DB2SQL
5    WHEN (wichtigkeit (altezeile.jobcode, altezeile.projekt) > 20)
6    SIGNAL SQLSTATE '70010' ('Wir brauchen diese Person');
7

```

Listing 5.3: Keine Entlassung wichtiger Personen

```

1  CREATE TRIGGER temp_aenderung
2    AFTER UPDATE ON Temperaturen
3    REFERENCING NEW AS neuezeile
4    FOR EACH ROW MODE DB2SQL
5    WHEN (neuezeile.temp >
6          (SELECT maxtemp FROM extremwerte
7           WHERE ort = neuezeile.ort)
8          OR
9          (SELECT maxtemp FROM extremwerte
10            WHERE ort = neuezeile.ort) IS NULL)
11   UPDATE Extremwerte
12     SET maxtemp = neuezeile.temp,
13         maxdate = CURRENT DATE
14     WHERE ort = neuezeile.ort;

```

Listing 5.4: Automatisches Anpassen einer Extremwerttabelle

```

1  CREATE TRIGGER neue_buchung
2    AFTER INSERT ON Buchungen
3    REFERENCING NEW_TABLE AS neuezeile
4    FOR EACH STATEMENT MODE DB2SQL
5    INSERT INTO Protokoll (Wann, Durchwen, AnzZeilen)
6      VALUES (CURRENT TIMESTAMP, USER,
7              (SELECT COUNT(*) FROM neuetab));

```

Listing 5.5: Protokollieren von Kontenbuchungen

5.10 Anwendungsentwicklung

SQL alleine reicht für die Erstellung von Anwendungen meist nicht aus. Daher werden Spracherweiterungen und Bibliotheken für Programmiersprachen zur Verfügung gestellt, die einen Zugriff auf die Datenbank über SQL gestatten. Die eigentliche Anwendungsentwicklung kann auf unterschiedlichen Methoden basieren, die alle ihre Vor- und Nachteile haben:

- ▶ Embedded SQL (statisch und dynamisch)
- ▶ Call Level Interface (CLI, ODBC)
- ▶ JAVA-Schnittstellen (JDBC, SQLJ)
- ▶ Direkte Programmierschnittstellen (Native API)
- ▶ Microsoft Data Objects (DAO, RDO, ADO)
- ▶ Erweiterung von SQL um Kontrollstrukturen
- ▶ Schnittstellen von Drittanbietern

5.10.1 Embedded SQL

Embedded SQL erlaubt die direkte Integration von SQL-Anweisungen in den Quelltext. So erstellte Programme werden als Host-Programme bezeichnet, deren Basis die Host-Sprache ist. Mögliche Host-Sprachen sind C/C++, Java (SQLJ), Cobol, Fortran und REXX.

Embedded SQL kann sowohl für statische als auch dynamische SQL-Anweisungen verwendet werden. Im statischen Fall sind die SQL Anweisungen sowie die referenzierten Objekte wie Tabellen und Spalten bei der Programmerstellung bekannt. Nur Vergleichswerte einer Abfrage oder Werte für Änderungen können variabel sein. Insbesondere müssen die genutzten Objekte in der Datenbank schon existieren. Bevor ein ausführbares Programm erstellt werden kann, muss der Quelltext von einem Precompiler vorübersetzt werden. Dieser erzeugt aus dem Host-Programm eine Quelltextdatei, die von einem Standardcompiler der Host-Sprache übersetzt werden kann, sowie ein Paket (Package). Dieses Paket enthält alle übersetzten SQL-Anweisungen und muss in der Datenbank abgelegt (gebunden) werden. Dabei werden die SQL-Anweisungen für den aktuellen Datenbestand optimiert. Das fertige Programm ruft bei Datenbankzugriffen dann die Elemente im Paket in der Datenbank auf, es werden keine SQL-Anweisungen mehr für den Datenbankzugriff benötigt, da diese schon in der Datenbank bekannt und übersetzt sind. Dies hat sehr große Performancevorteile, zur Laufzeit entfällt die Analyse und das Optimieren von SQL-Anweisungen. Ändert sich der Datenbestand derart, dass ein anderer Ausführungsplan optimal ist, muss das Paket jedoch neu optimiert werden.

Dynamisches Embedded SQL erlaubt die Ausführung von SQL-Anweisungen, die zur Zeit der Programmerstellung noch nicht bekannt sind, da beispielsweise der Benutzer des Programms diese selbst in das Programm eingibt. Die Anfrageoptimierung findet dann erst zur Laufzeit bezüglich des aktuellen Datenbestands *dynamisch* statt. Nachteilig wirkt sich diese Methode jedoch auf die Performance aus.

5.10.2 Call Level Interface und ODBC

Das Call Level Interface (CLI) ist eine Schnittstelle für die Entwicklung unter C und C++. Die Schnittstelle basiert dabei auf Microsofts Open Database Connectivity Standard (ODBC) sowie dem X/Open und ISO CLI-Standard. Dies erlaubt eine einfache Einarbeitung für Entwickler, die mit diesen Standards vertraut sind, sowie eine einfache Portierung existierender Programme. Die meisten ODBC-Programme können sogar ohne Anpassung direkt mit anderen Datenbanksystemen eingesetzt werden.

Die SQL-Integration ist dynamisch, wodurch sich die gleichen Vor- und Nachteile wie bei dynamischen embedded SQL ergeben. Es wird jedoch kein Precompiler benötigt, die SQL-Anweisungen werden als Parameter an Schnittstellenfunktionen weitergegeben. Auch ein Binden von Paketen für eine Anwendung entfällt.

5.10.3 JAVA-Schnittstellen (JDBC, SQLJ)

Sowohl JDBC¹⁵ (CLI-Schnittstelle) als auch SQLJ (Embedded SQL für JAVA) sind herstellerunabhängige Standardprogrammierschnittstellen für JAVA. Die Schnittstellen können dabei sowohl für Anwendungsprogramme, Applets und Servlets eingesetzt werden. Dadurch vereinigen diese Sprachintegrationen die Vorteile der Datenbankschnittstellen mit denen von JAVA.

Die Vor- und Nachteile bei JDBC und SQLJ sind vergleichbar zu ODBC und embedded SQL. Insbesondere benötigt SQLJ einen Precompiler, der SQLJ-Quelltexte in reinen JAVA-Quelltext überführt und ein Paket erstellt, das in der Datenbank gebunden werden muss.

¹⁵Die Bezeichnung JDBC wird häufig als Abkürzung für Java Database Connectivity angegeben, obwohl JDBC kein Akronym, sondern ein geschützter Name ist.

SQLJ erlaubt die Integration statischer Anweisungen mit Vorübersetzung und Optimierung in Form von Paketen. Der eigentliche Zugriff auf die Datenbank und die Pakete geschieht jedoch über die JDBC-Schnittstelle.

5.10.4 Direkte Programmierschnittstellen (Native APIs)

Datenbanken wie DB2 bieten direkte Programmierschnittstellen für die Verwaltung von Instanzen und Datenbanken, die von C/C++, Cobol und Fortran eingesetzt werden können. Diese Schnittstellen werden deshalb auch als administrative Schnittstellen bezeichnet. Die Schnittstellen sind im Kern Funktionsbibliotheken, so dass ein Precompiler nicht erforderlich ist. Informationen werden über eigens definierte Datenstrukturen ausgetauscht. Ein Beispiel hierfür ist das Oracle Call Interface (OCI).

5.10.5 Microsoft Data Objects (ADO, DAO, RDO)

Für die Anwendungsentwicklung unter Visual Basic und Visual C++ hat Microsoft die Schnittstellen *Data Access Objects* (DAO) und *Remote Data Object* (RDO) entwickelt, die auf ODBC basieren. Die neueste Schnittstelle von Microsoft ist *ActiveX Data Object* (ADO), die sich von ODBC löst und eine objektorientierte Schnittstelle bietet. ADO zeichnet sich durch seine hohe Geschwindigkeit, den einfachen Einsatz sowie den geringen Speicheroverhead aus. Mit der Verwendung einer dieser Schnittstellen ist man jedoch an Windows als Client-Plattform gebunden.

5.10.6 Erweiterung von SQL um Kontrollstrukturen

SQL wurde als eine interaktive Sprache entworfen, bei der jede Anweisung eine Einheit bildet, die sofort ausgeführt wird. Der Hersteller Oracle hat SQL in seinem Produkt um Kontrollstrukturen wie Schleifen und bedingte Anweisungen erweitert und damit die Möglichkeit geschaffen, direkt mit SQL zu programmieren. Der Name dieser Sprache ist bei Oracle PL/SQL, im SQL3-Standard ist eine solche prozedurale Spracherweiterung ebenfalls vorgesehen.

Diese Spracherweiterungen erlauben dabei auch die Definition von Funktionen und Prozeduren (Stored Procedures). Diese werden dabei vom DBMS ausgeführt, das immer die Kontrolle über die Befehlausführung hat. Mit diesem Konzept kann neben den Datendefinitionen auch *Funktionalität* der modellierten Anwendung redundanzfrei im DBMS kontrolliert verwaltet werden. Die wichtigsten Vorteile der Stored Procedures sind:

- ▶ Wie in Programmiersprachen sind Funktionen und Prozeduren ein bewährtes Strukturierungsmittel für größere Anwendungen.
- ▶ Die Angabe der Funktionen und Prozeduren erfolgt in der Datenbanksprache selbst und sind damit nur vom DBMS und nicht von der Programmiersprache oder der Betriebssystemumgebung abhängig.
- ▶ Das DBMS kann eine Optimierung der Prozeduren vornehmen.
- ▶ Die Prozedurausführung erfolgt vollständig unter Kontrolle des DBMS. Dies bringt insbesondere Vorteile in Client/Server-Architekturen: Die Prozedur kann im Server ausgeführt werden, ohne dass Daten oder Prozedurcode über das Netz geschickt werden müssen.
- ▶ Die zentrale Kontrolle der Prozeduren ermöglicht eine redundanzfreie Darstellung relevanter Aspekte der Anwendungsfunktionalität.
- ▶ Konzepte und Mechanismen der Rechtevergabe des DBMS können auf Prozeduren erweitert werden.

- ▶ Prozeduren können in der Integritätssicherung beispielsweise im Triggeraktionsteil verwendet werden.

Neben reinen SQL-Routinen ist es bei einigen Systemen auch möglich, Prozeduren in C, Cobol, Fortran, Java und anderen Programmiersprachen zu entwickeln, die auch auf dem Server ausgeführt werden, so dass einige der Vorteile auch ohne SQL-Erweiterung ermöglicht werden. Diese Variante ist auch meist mit einer besseren Performance der Prozeduren verbunden.

5.10.7 Schnittstellen von Drittanbietern

Diverse Drittanbieter bieten Werkzeuge für die Anwendungsentwicklung an. Diese basieren auf den Standardschnittstellen der jeweiligen Datenbank, stellen jedoch dem Benutzer eine weitere Schnittstelle zur Verfügung, die beispielsweise einfacher von der Handhabung ist oder eine grafische Schnittstelle bietet. Beispiele sind

- ▶ Datenbankentwicklungstools wie Lotus Approach oder Microsoft Access, die auf ODBC aufbauen. Anwendungen werden hier mit Hilfe von Assistenten teilautomatisiert erstellt.
- ▶ Net.Data unterstützt die Erstellung von Web-Anwendungen.
- ▶ PERL DBI ist eine Schnittstelle für die Skriptsprache Perl.

Kapitel 6

Objektrelationale Konzepte

In der Software-Entwicklung haben objektorientierte Konzepte sich seit langem bewährt. Das relationale Modell hat sich als adäquate Möglichkeit zur Darstellung von Daten erwiesen. Das relationale Datenbankmodell bringt hinsichtlich einer einfachen Verbindung dieser Welten jedoch schon auf Ebene der Anfragesprache eine Reihe von Schwachstellen mit:

Strukturmangel im Anfrageergebnis:

Das Ergebnis einer Anfrage in SQL ist eine Relation. Wird bei der Anfrage ein Join durchgeführt, werden dadurch Informationen redundant mit zusammenhängenden Datensätzen zurückgegeben.

Keine Unterstützung komplexer Strukturen

Notwendigkeit expliziter Verbundoperationen:

Zur Vermeidung von Redundanzen bei der Datenhaltung werden Informationen auf verschiedene Tabellen verteilt, die dann bei Anfragen wieder explizit mittels einem Join zu verbinden sind. Dies muss bei jeder Anfrage vom Entwickler explizit formuliert werden, wozu insbesondere die notwendigen Fremdschlüssel bekannt sein müssen.

Änderungsoperationen:

Sind bei einer Änderung Objekte aufgrund einer Eigenschaft zu ändern, die in einer eigenen Relation gespeichert sind, so ist beim Update-Befehl in der WHERE-Bedingung immer eine Unterabfrage erforderlich, um den Bezug zu der Information herzustellen, über die die Bedingung formuliert werden kann.

Zur Beseitigung dieser Probleme bietet es sich daher an, die bekannten objektorientierten Möglichkeiten mit dem vertrauten Relationenmodell zu verbinden. Insbesondere können die bisherigen Datenbankkenntnisse dann auch weiter verwendet werden und die objektorientierten Konzepte der Anwendungsmodellierung auf die Datenbankwelt übertragen werden.

Für eine nähere Begriffsbestimmung objektrelationaler DBMS fehlt in der Literatur bis dato eine allgemein anerkannte Definition. Infolgedessen orientiert man sich an den Konzepten des SQL-Standards, um den Begriff Objektrelationalität einzugrenzen. Ein Grundelement objektrelationaler Systeme ist die Möglichkeit, Strukturen für komplexe Objekte zu erzeugen. Benutzerdefinierte Strukturen und benutzerdefinierte Funktionen erlauben es, eigene Datentypen zu definieren und diese analog zu den Grunddatentypen zu verwenden. Zusätzlich können bei Aufbau dieser Strukturen objektorientierter Elemente wie der Vererbung eingesetzt werden. Die so definierten Klassen einer Klassenhierarchie können dann direkt in Tabellen umgesetzt werden und mit Methoden versehen werden. Diese Bereiche wurden auch in den SQL-Standard aufgenommen. So umfasst SQL:1999 folgende objektorientierte Erweiterungen.

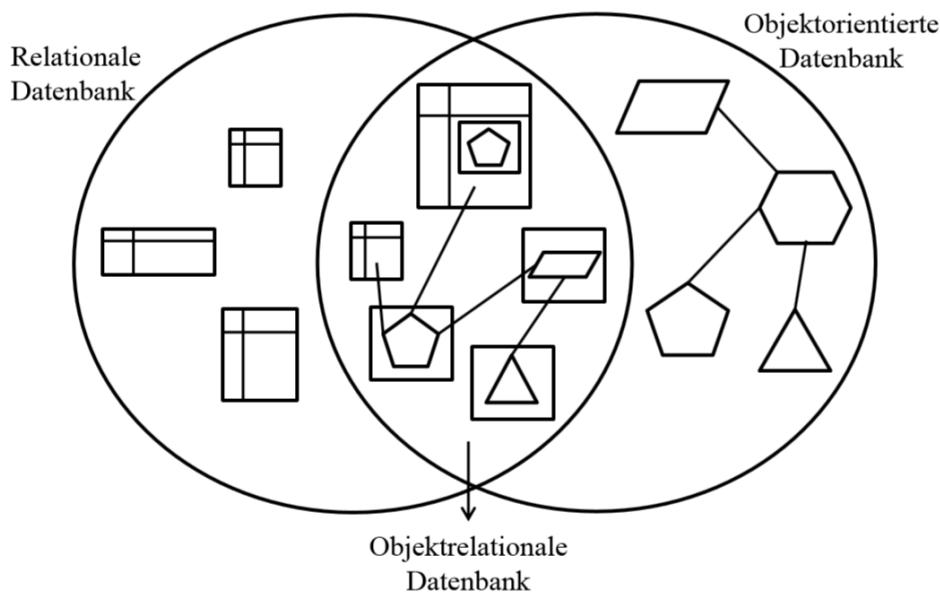


Abbildung 6.1: Synthese von relationalen und objektorientierten Konzepten

- ▶ Objektidentifikationen
- ▶ Basisdatentypen BLOB (Binary Large Object) und CLOB (Character Large Object)
- ▶ Typkonstruktoren für die Definition von benutzerdefinierten Datentypen
- ▶ Typisierte Tabellen (so genannte Objekttabellen) mit der Möglichkeit der Subtabellenbildung
- ▶ Typisierte Sichten (erweitertes Viewkonzept)

Mit SQL:2003 kamen weitere Ergänzungen hinzu:

- ▶ Typkonstruktor für Mehrfachmengen
- ▶ Tabellenwertige Funktionen
- ▶ Spalten zur Generierung künstlicher Schlüssel
- ▶ Spalten, deren Werte von anderen Spalten abgeleitet sind
- ▶ Verknüpfung zwischen SQL und XML

6.1 Kollektionen

Das klassische Relationenmodell verlangt, dass die Attributwerte der Tabellen atomar sind und keine zusammengesetzten Werte ausdrücken. SQL:1999 wurde nun dahingehend erweitert, dass mittels Typkonstruktoren weitere Datentypen definiert werden können.

Kollektionen ermöglichen mehrwertige Attribute wie beispielsweise Telefonnummern oder Email-Adressen eines Mitarbeiters. Hierfür sind prinzipiell unterschiedliche Kollektionsarten vorstellbar. So können Kollektionen geordnet oder ungeordnet sein, sie dürfen Duplikate enthalten oder nicht und sie können von beliebiger oder beschränkter Länge sein. Dies kann durch folgende Typkonstruktoren definiert werden:

ARRAY: Arraytypkonstruktor

MULTISET: Multimengentypkonstruktor¹

SET: Mengentypkonstruktor

LIST: Listentypkonstruktor

6.1.1 Arrays

Arrays sind geordnete Kollektionen mit einer im Schema zu definierenden Maximalkardinalität:



Ein beliebiger Datentyp (außer ein Array) kann als Sammlung durch das Schlüsselwort **array** erweitert werden, indem in eckigen Klammern die Anzahl möglicher Elemente in dieser Sammlung spezifiziert werden. Damit lassen sich mehrere Attributwerte als Einheit behandeln. Dies hat den Vorteil, dass bei einer Anfrage direkt diese Kollektion verwendet werden kann, ohne auf weitere Tabellen zugreifen zu müssen.

Ein Beispiel:

```
CREATE TABLE Umsatz (
    Produkt      char(20),
    Monatsbetrag NUMBER(10,2) ARRAY [12],
    ...)
```

Zur Erzeugung eines Arrays stehen folgende Möglichkeiten zur Verfügung:

- ▶ **ARRAY[]:** Generiert ein leeres Array.
- ▶ **ARRAY[Werteliste]:** Generiert ein Array, der mit den Werten gefüllt wird. Die Anzahl der Elemente kann dabei auch kleiner als die Maximallänge sein.
- ▶ **ARRAY(Anfrage):** Erzeugt einen Array aus einer einspaltigen SQL-Anfrage.

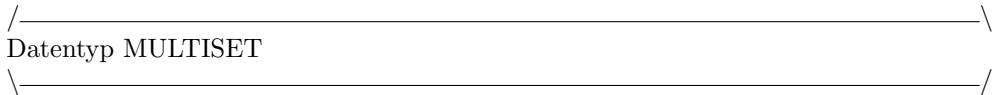
Mittels **[index]** wird auf das Element mit dem angegebenen Index zugegriffen. Der erste Arrayeintrag hat dabei den Index 1! Die Funktion **CARDINALITY(Array)** liefert die Anzahl der Elemente eines Arrays. Auch können Arrays mit dem Konkatenationsoperator **||** miteinander verbunden werden.

Zwei Arrays sind vergleichbar, wenn ihre Elementtypen vergleichbar sind. Zwei vergleichbare Arrays sind gleich, wenn sie dieselbe Kardinalität besitzen und alle Elemente paarweise gleich sind. NULL-Werte führen in bekannterweise zum Wahrheitswert UNKNOWN.

¹Bei Oracle heißt der Multimengentypkonstruktor TABLE statt MULTISET.

6.1.2 Multimengen

Multimengen können analog zu Arrays direkt beim CREATE TABLE erzeugt werden:



Ein Beispiel:

```

CREATE TABLE Buch (
    InvNr          char(10),
    Autoren        varchar(5) MULTISSET,
    ...
)

```

Zur Erzeugung einer Multimenge stehen folgende Möglichkeiten zur Verfügung:

- ▶ **MULTISET[]**: Generiert eine leere Multimenge.
- ▶ **MULTISET[Werteliste]**: Erzeugt aus einer Werteliste eine Multimenge.
- ▶ **MULTISET(Anfrage)**: Erzeugt eine Multimenge aus einer einspaltigen SQL-Anfrage.

Ein Beispiel:

```

INSERT INTO Buch (InvNr, Autoren) \newline
\quad \quad VALUES (MULTISET ('Skaja Müller', 'Peter Maier'))

```

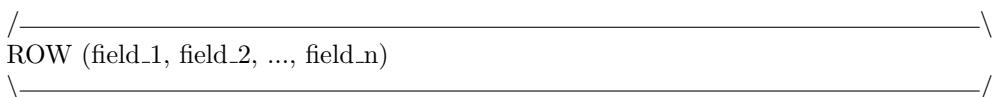
Auch hier liefert die Funktion **CARDINALITY** die Kardinalität der Multimenge. Mittels **Wert MEMBER mm** kann getestet werden, ob ein Wert in einer Multimenge enthalten ist. **mm1 SUBMULTISET mm2** dient zur Überprüfung, ob eine Teilmengenbeziehung vorliegt. Auch stehen die bekannten Mengenfunktionen **UNION**, **INTERSECT** und **EXCEPT** zur Verfügung.

Mittel **mm IS A SET** kann überprüft werden, ob es sich bei der Multimenge um eine echte Menge handelt. Mittels **SET mm** kann eine Multimenge in eine Menge überführt werden, alle Duplikate werden entfernt.

Zwei Multimengen sind vergleichbar, wenn die Elementtypen vergleichbar sind. Zwei vergleichbare Multimengen sind genau dann gleich, wenn sie die gleichen Elemente mit der jeweils gleichen Kardinalität enthalten.

6.2 Zeilentypen

Neben Kollektionen gibt es die Möglichkeit, ein Attribut selbst als strukturierten Datentyp vorzusehen. Dies geschieht mittels Zeilendatentypen (row types), ausgezeichnet mit dem Schlüsselwort **row**.



Die einzelnen Felder dieses Zeilendatentyps werden analog zur Attributangabe beim CREATE TABLE durch Feldname, Felddatentyp und Feldoption festgelegt.

Ein Beispiel:

```
create table niederlassung (
    id          number(10) primary key,
    name        varchar(30),
    adresse     row (
        strasse  varchar(30),
        ort      varchar(30),
        plz      char(5))
    ansprechpartner number(10) array[10]
)
```

Neben diese direkt beim `create table` angebbaren Definition des Zeilentyps kann dieser auch zunächst explizit über den Befehl `create row type` definiert und dann mehrfach innerhalb Tabellendefinitionen verwendet werden:

```
create row type rt_adresse (
    strasse  varchar(30),
    ort      varchar(30),
    plz      char(5));

create table niederlassung (
    id          number(10) primary key,
    name        varchar(30),
    adresse     rt_adresse
    ansprechpartner number(10) array[10]
)
```

Der Befehl `ROW (Wert, Wert, ...)` erzeugt einen strukturierten Wert beliebigen Aufbaus. Die explizite Definition eines Zeilentyps mit eigenem Namen erlaubt dagegen über Typ `(Wert, Wert, ...)` einen zu einem Typ passenden strukturierten Wert zu erzeugen. Der Befehl `create row type` wird jedoch vielfach nicht unterstützt.

Mittels des Punkt-Operators kann analog zur Programmiersprache C auf die Elemente einer Struktur zugegriffen werden:

```
select adresse.ort
    from niederlassung
    where name = 'Mango'
```

Strukturierte Werte sind auch mittels der üblichen Vergleichsoperatoren vergleichbar. Hierfür müssen die Werte dieselbe Anzahl an Feldern besitzen und alle Felder paarweise vergleichbare Datentypen besitzen. Bei <- und >-Vergleichen wird lexikographisch verglichen, NULL-Werte führen in bekannter Weise zum Wahrheitswert UNKNOWN.

6.3 Strukturierte Datentypen - Typkonstruktoren

Die hinzugekommenen Möglichkeiten können nun nicht nur innerhalb von Tabellendefinitionen verwendet werden, es besteht die Möglichkeit benutzerdefinierte Typen als eigenständige Datenbankelemente vergleichbar zu dem Klassenbegriff der Objektorientierung zu definieren.

Ein strukturierter Datentyp besteht wie Tabellen aus einer Folge von Attributen, die jeweils einen Datentyp haben müssen. So definierte Typen können in einer Tabellendefinition, so genannte typisierte Tabellen, überführt werden. Ein strukturierter Typ kann dabei ein Untertyp (Subklasse) eines schon existierenden Typs sein, wobei er dann alle Eigenschaften (Attribute) des Obertyps (Superklasse) erbt. Diese Hierarchiebildung kann dabei über mehrere Stufen hinweg erfolgen, die daraus resultierenden Tabellen ergeben dann eine Hierarchie typisierter Tabellen. Strukturierte Datentypen und typisierte Tabellen erlauben dadurch eine bessere Datenmodellierung, das den objektorientierten Vererbungsmechanismus integriert. Die Gesamtsyntax hat dabei folgenden Aufbau:

```

CREATE TYPE <UDT name> [<subtype clause>] [AS <representation>]
    [<instantiable clause>] <finality> [<reference type
        specification>] [<cast option>] [<method specification list>]
<subtype clause> ::= UNDER <supertype name>
<represenation> ::= <predefined type> | [( <member> , ...)]
<instantiable clause> ::= INSTANTIABLE | NOT INSTANTIABLE
<finality> ::= FINAL | NOT FINAL
<member> ::= <attribute definition>
<reference type specification> ::= REF USING <predifined type> |
    REF FROM (<list of attributes>) |
    REF IS SYSTEM GENERATED
<method specification list> ::= <original method spcification> |
    <overriding method specification>
<original method specification> ::=
    <partial method specification> <routine characteristics>
<overriding method specification> ::=
    OVERRIDING <partial method specification>
<partial method specification> ::= [INSTANCE | STATIC] METHOD
    <routine name> <SQL parameter declaration list> <return clause>

```

Nachfolgend werden einzelne Bereiche detaillierter betrachtet.

6.3.1 Objektidentifikatoren und Referenzen

Das Konzept der Objektidentifikatoren (OIDs) ermöglicht eine zustandsunabhängige, unveränderbare Identifikation von Datenbankobjekten. Sie hat gegenüber dem Primärschlüssel den Vorteil, eine stabile Referenz zu sein, die sich nie ändert. Das Konzept der OIDs bildet darüber hinaus die Grundlage für die Verwendung von Pfadausdrücken und vereinfacht dabei die Formulierung diverser Datenbankanfragen.

Beim Einsatz strukturierter Datentypen als Typ einer Tabelle wird jede Zeile dieser Tabelle mit einer OID assoziiert. Je nach Umsetzung ist die OID sichtbar als zusätzliche Spalte der Tabelle dargestellt oder unsichtbar als gekapselte Eigenschaft der Instanz realisiert werden. Im letzteren Fall kann die OID über eine spezielle Funktion ermittelt werden.

Meist werden OIDs durch das DBMS generiert. Der SQL-Standard erlaubt jedoch auch eine benutzerdefinierte Festlegung der OIDs, so dass der Programmierer beim Erzeugen eines Datensatzes auch für die Erzeugung der OID verantwortlich ist. Insbesondere ist dies im Zusammenhang mit objektorientierten Sichten für stabile Objektreferenzen von Bedeutung (vgl. Abschnitt 6.5). OIDs sind damit im Vergleich zu Objektorientierten Datenbanken nicht systemweit eindeutig sondern nur innerhalb eines Objekttyps eindeutig.

6.3.2 Definition strukturierter Datentypen

Zum Anlegen eines strukturierten Typs dient der Befehl `CREATE TYPE`. Die AS-Klausel enthält dabei die mit dem Typ verbundenen Attribute, die analog zum `CREATE TABLE` angegeben werden. Mögliche Datentypen für die Attribute sind dabei neben den Standarddatentypen auch die zuvor vorgestellten Kollektionstypen und Zeilentypen. Auch andere mit `CREATE TYPE` angelegte Typen können in Typdefinitionen verwendet werden. Die UNDER-Klausel erlaubt die Angabe einer Oberklasse, wodurch automatisch alle Attribute der Oberklasse vererbt werden.

```
/-----\
CREATE TYPE typname [UNDER Supertyp] AS (
    Attributdefinitionsliste
)
[[NOT] INSTANTIABLE]
[[NOT] FINAL]
[REF IS SYSTEM GENERATED | FROM (Attributliste) | USING Typ]
[Methoden]
\-----/
```

In SQL:1999 werden Objektidentifikationen (OIDs) weitreichend unterstützt. Eine OID ist dabei eine Eigenschaft einer Instanz, die sie eindeutig identifiziert und sie von allen anderen Instanzen des gleichen Typs oder anderer Typen unterscheidet. Der Standard legt kein festes Format für OIDs fest, dieses wird bei den Typdefinitionen angegeben. Dabei können sie aus Attributen gebildet werden (vergleichbar zum bekannten Primärschlüssel), auch die Bildung künstlicher OIDs ist möglich. Werden Identifikatoren nicht auf Attribute eines Typs zurückgeführt, dann kann festgelegt werden, ob ihre Werte benutzerdefiniert oder systemdefiniert sind.

Die Klausel `REF USING` definiert die Grundeigenschaften der OID. Man erkennt am Namen, dass OIDs insbesondere für Referenzen auf Objekte (vergleichbar zu Fremdschlüssel) nützlich sind. Die Angabe ist dabei nur bei Wurzelklassen notwendig um Referenzen auf die Klassenelemente der Ober- und aller Unterklassen zu ermöglichen. Wird ein strukturierter Typ in eine Tabelle überführt, wird automatisch eine Spalte OID mit dem hier angegebenen Datentyp erzeugt. Mögliche Typen sind dabei `SIMALLINT`, `INTEGER`, `BIGINT`, `DECIMAL`, `CHAR` und `VARCHAR`.

Alternativ kann auch `REF IS SYSTEM GENERATED` verwendet werden, wobei dann das System selbst die OID-Vergabe steuert. Dies ist der Default. Möchte man ein Attribut der Struktur als OID verwenden, ist `REF FROM Spaltenname` anzugeben. Untertypen besitzen keine Angabe zur OID, dies wird vom Supertyp immer übernommen.

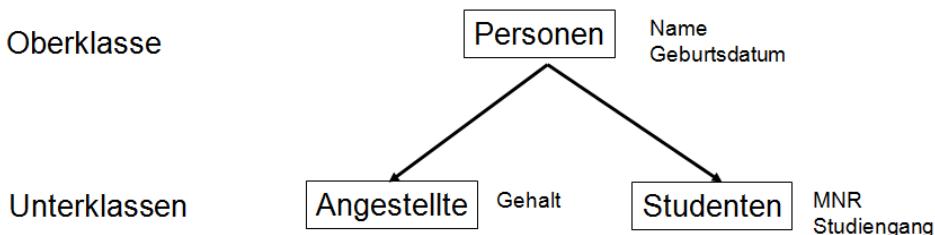


Abbildung 6.2: Beispiel einer Vererbungshierarchie

```
CREATE TYPE Personen_T
AS (Name VARCHAR(20),
    Geburtsdatum DATE)
REF USING INTEGER;
```

```

CREATE TYPE Angestellte_T UNDER Personen_T
AS (Gehalt INT);

CREATE TYPE Studenten_T UNDER Personen_T
AS (MNR CHAR(6),
Studiengang VARCHAR(30));

```

Der optionale Zusatz `final` am Ende der Typdefinition verhindert, dass von dem Typ neue Untertypen gebildet werden können. Ein weiterer möglicher Zusatz ist `not instantiable`, mit dem der Typ als abstrakt gekennzeichnet wird. In diesem Fall können keine Instanzen dieses Typs gebildet werden.

Nach einer Typdefinition können mit `ALTER TYPE` Spalten gelöscht und hinzugefügt werden.

Beispiele:

```

⇒ ALTER TYPE Personen_T ADD ATTRIBUTE Tel CHAR(12)
⇒ ALTER TYPE Personen_T DROP ATTRIBUTE Tel

```

6.3.3 Methoden

Jeder instanzierbare strukturierte Typ besitzt einen Defaultkonstruktor, der eine Instanz des strukturierten Typs mit Defaultbelegungen erzeugt. Der Defaultkonstruktor ist eine parameterlose Funktion, die genauso heißt wie der zugehörige strukturierte Typ.

Für jedes Attribut werden automatisch zwei Methoden generiert, die den Wert des Attributs lesen bzw. ändern. Diese beiden Methoden heißen wie das jeweilige Attribut. Die parameterlose Methode gibt dabei den Wert zurück. Die Methode mit einem Parameter belegt das Attribut mit dem übergebenen Wert.

Die Möglichkeit, klassen- oder typspezifisches Verhalten definieren und implementieren zu können, ist eine grundlegende objektorientierte Eigenschaften und muss deshalb auch in objektrelationalen Modellen berücksichtigt werden. Da die Methodenimplementierung sehr systemspezifisch ist, wird hier nicht näher auf diese Thematik eingegangen.

6.3.4 Erzeugen typisierter Tabellen

Auf Basis strukturierter Typen können direkt Tabellen (so genannte typisierte Tabellen) erstellt werden, wobei auch die durch Vererbung hinzugekommenen Attribute berücksichtigt werden. Tabellen ohne Obertabelle werden dabei als Wurzeltablelle bezeichnet. Mit der Trennung des Typs und der Extension einer Tabelle findet auch die Trennung der Typ- und Tabellenhierarchien statt. Eine Tabellenhierarchie ist jedoch von einer Typhierarchie derart abhängig, dass die Typen der Instanzen der Subtabellen Subtypen der Typen der Instanzen der Supertabelle sein müssen.

```

/-----\
CREATE TABLE tabname OF typename [UNDER Supertabname]
(
  REF IS oid-spalte [SYSTEM GENERATED | DERIVED | USER GENERATED]
  [Integritätsbedingungen]
)
\-----/

```

Die REF-IS-Klausel legt den Namen der OID-Spalte fest. Die Referenzgenerierungsart wird durch die Referenztypspezifikation des zugrundeliegenden Typs bestimmt und kann hier mit aufgeführt, jedoch nicht umgestellt werden.

Das Beispiel:

```

CREATE TABLE Personen OF Personen_T
(REF IS OID USER GENERATED);

CREATE TABLE Angestellte OF Angestellte_T UNDER Personen
INHERIT SELECT PRIVILEGES;

CREATE TABLE Studenten OF Studenten_T UNDER Personen
INHERIT SELECT PRIVILEGES;

```

Die OID-Angabe muss nur für die Wurzeltabellen angegeben werden, da auch nur diese in ihrer zugehörigen Typ-Definition eine REF-Klausel enthält. Die OIDs sind für die Datensätze in einer Tabelle eindeutig und werden beim Einfügen angegeben (USER GENERATED). OID-Werte können nicht verändert werden. Der Zusatz INHERIT SELECT PRIVILEGES hat zur Folge, dass jeder Benutzer, der lesenden Zugriff auf die Oberklasse hat, auch die Datensätze der Unterklassen lesen darf.

6.3.5 Löschen typisierter Tabellen

Das Löschen typisierter Tabellen geschieht wie das Löschen von Standardtabellen über den Befehl DROP TABLE. Eine Tabelle kann dabei nur gelöscht werden, wenn sie keine Untertabellen besitzt. Das Löschen einer vollständigen Tabellenhierarchie kann jedoch in einen Befehl zusammengefasst werden.

```
/-----\
DROP TABLE HIERARCHY wurzeltable
\-----/
```

6.3.6 Einfügen von Zeilen

Auch bei typisierten Tabellen werden Zeilen über den INSERT-Befehl eingefügt.

```

INSERT INTO Personen (oid, name, geburtstag)
VALUES (Personen_T(10), 'Hannah', '10.01.2001');

INSERT INTO Angestellte (oid, name, geburtstag, Gehalt)
VALUES (Angestellte_T(20), 'Max', '10.01.1950', 100000);

INSERT INTO Studenten (oid, name, geburtstag, MNR, Studiengang)
VALUES (Studenten_T(30), 'Willy', '10.01.1952', 10, 'WI');

```

Für die Spalte OID ist immer die Verwendung einer Casting-Funktion erforderlich, da diese intern immer den Typ REFERENCE hat. Die Casting-Funktionen werden automatisch erzeugt, wobei der Argumenttyp dem Typ, der beim CREATE TYPE angegeben wurde, entspricht.

6.3.7 Weitere DML-Operationen

Eine Selektion auf eine typisierte Tabelle liefert alle Datensätze der Tabelle und aller Untertabellen, die der WHERE-Bedingung entsprechen, da Elemente einer Untertabelle auch automatisch zu der allgemeineren Obertabelle gehören. So liefert eine Selektion auf die Personentabelle auch Angestellte und Studenten. Möchte man nur die Datensätze haben, die nur in der Obertabelle und in keiner Untertabelle enthalten sind, kann man dies über den Zusatz ONLY erreichen.

Beispiel:

```
⇒ SELECT * FROM ONLY(Personen)
```

Der Zusatz ONLY kann auch beim UPDATE und DELETE Befehl verwendet werden, wodurch nur die Einträge auf der jeweiligen Ebene verändert werden.

Für die Selektion aller Datensätzen einschließlich aller Attribute, die in der Vererbungshierarchie auftauchen, ist der Zusatz OUTER zu verwenden. Dabei wird bei Datensätzen, die aufgrund der Vererbung über ein Attribut nicht verfügen, ein NULL-Wert eingetragen. Insbesondere verdeutlicht dies die interne Speicherung der Vererbungshierarchie in einer Überrelation.

Beispiel:

```
⇒ SELECT * FROM OUTER(Personen)
```

Ausgewählte Subtabellen können mit Hilfe des Mengenoperators EXCEPT CORRESPONDING ausgeschlossen werden:

Beispiel:

```
⇒ SELECT * FROM Personen EXCEPT CORRESPONDING TABLE Studenten
```

Wird die OID in der WHERE-Klausel verwendet, ist auch hier der Einsatz der Casting-Funktionen erforderlich.

Beispiel:

```
⇒ UPDATE Personen
    SET Name = 'Mueller'
    WHERE OID = Personen_T(10)
```

6.3.8 Physische Darstellung typisierter Tabellen bei DB2

In der Datenbank befindet sich für eine Tabellenhierarchie immer nur eine Tabelle (die H-Tabelle), in der die Daten aller Tabellen der Hierarchie abgelegt sind. Insbesondere hat diese Tabelle auch alle möglichen Spalten, wodurch viele NULL-Werte in einer Zeile auftauchen. Der Tabellenname entspricht dem Namen der Obertabelle mit dem Anhang _HIERARCHY. Dies kann beim Anlegen der Tabelle aber auch mit der Option HIERARCHY selbst definiert werden.

Die erste Spalte dieser Tabelle entspricht der TypID, über die das System erkennt, zu welchem Typ ein Eintrag gehört. Zur Ermittlung des Typnamens für eine ID kann auf das Data Dictionary zugegriffen werden:

Beispiel:

```
⇒ SELECT TYPENAME, SOURCENAME, METATYPE
    FROM SYSCAT.DATATYPES
```

Die Spalte METATYPE in R für strukturierte Typen, U für typisierte Tabellen und W für typisierte Views. Informationen zur Hierarchie finden sich in der Tabelle SYSCAT.HIERARCHIES.

6.4 Referenzspalten

Beziehungen zwischen strukturierten Typen können direkt innerhalb der Typdefinition hinterlegt werden, indem Referenzspalten in die AS-Klausel aufgenommen werden. Referenzwerte werden intern als OID-Werte gespeichert, sie erlauben jedoch einen im Vergleich zu Fremdschlüsseljoins komfortableren Zugriff auf andere referenzierte Objekte.

Der SQL:1999-Standard führt hierfür den Referenztyp ein. Referenztypen können als Wertebereich in Typdefinitionen verwendet werden. Referenzierte Tupel sind 'teilbar', d.h. es kann beliebig viele Referenzen auf ein bestimmtes Objekt möglich. Dies ist ebenfalls ein großer Vorteil von Referenztypen im Vergleich zur direkten Verwendung eines vorhandenen Typs in einer Typdefinition. Zur Erzeugung eines Referenztyps ist das Schlüsselwort REF gefolgt vom Typnamen in Klammern anzugeben:

```

CREATE TYPE Abteilung_T
AS (Name CHAR(40),
    Ort CHAR(40))
REF USING INTEGER;

CREATE TYPE Angestellte_T UNDER Personen_T
AS (Gehalt INTEGER,
    Abteilung REF(Abteilung_T));

CREATE TABLE Abteilung OF Abteilung_T
REF IS OID USER GENERATED;

CREATE TABLE Angestellte OF Angestellte_T UNDER Personen_T
INHERIT SELECT PRIVILEGES
(Abteilung WITH OPTIONS SCOPE Abteilung)

```

Der Zusatz `WITH OPTIONS SCOPE Abteilung` weist das System hin, dass der Verweis auf die Tabelle Abteilung geht. Dies ist notwendig, da ausgehend von Typ `Abteilung_T` mehrere Tabellen erzeugt werden können. Dieser Verweis kann nun bei Selektionen in Verbindung mit dem Dereferenzierungsooperator `->` zur Herstellung einer Verbindung zwischen den Tabellen verwendet werden, ohne dass ein manueller Join erforderlich ist.

Beispiel:

```

⇒ SELECT A.name FROM Angestellte A
    WHERE A.Abteilung->Ort = 'Ravensburg'

```

Während mittels `->` der Zugriff auf einzelne Attribute möglich ist, kann mit der FUNKTION `DEREF` auf den vollständigen strukturierten Wert, auf den die Referenz verweist, zugegriffen werden.

Referenzspalten können auch direkt beim Anlegen einer Tabelle ohne vorherige Typdefinition für die anzulegende Tabeller genutzt werden:

```

CREATE TABLE Abteilungsleiter (
    PNR      NUMBER (10)
    Abteilung REF (Abteilung_T) SCOPE (Abteilung)
)

```

Die Belegung von Referenzspalten gestaltet sich schwieriger, da hier OIDs der referenzierten Objekte abzulegen sind, die insbesondere bei vom System vergebene OIDs zunächst zu ermitteln sind:

```

UPDATE Angestellter SET
    Abteilung = (SELECT OID From Abteilung WHERE Name = 'Forschung')
WHERE PNR = 66767

```

6.5 View-Hierarchien

Auch Views können auf Basis strukturierter Typen definiert werden.

```

CREATE VIEW Personen_V OF Personen_T
(REF IS VOID USER GENERATED)

```

```

AS SELECT Personen_T(INTEGER(OID)), Name, Geburtsdatum
      FROM ONLY(Personen);

CREATE VIEW Angestellte_V OF Angestellte_T
  UNDER Personen_V
  INHERIT SELECT PRIVILEGES
AS SELECT Angestellte_T(INTEGER(OID)), Name, Geburtsdatum,
         Gehalt FROM Angestellte;

```

Bei objektgenerierten Sichten liegt ein wesentliches Problem in der Gewährleistung der Stabilität von (beispielsweise über Joins oder Gruppierungen) generierten Objekten bzw. deren OIDs. Da der Inhalt von Sichten dynamisch mit jeder Anfrage neu berechnet wird, kann die Stabilität der OIDs nicht gewährleistet werden, wenn die OIDs jedes Mal wieder neu generiert werden müssten. Daher beschränken sich objektrelationale DBMS nur auf benutzergenerierte oder mittels Berechnung ableitbare OIDs für typisierte Sichten.

6.5.1 SQL-Funktionen für typisierte Tabellen und Views

Folgende Funktionen für typisierte Tabellen und Views stellt beispielsweise IBM DB2 zur Verfügung:

- ▶ DEREF
- ▶ TYPE_ID
- ▶ TYPE_NAME
- ▶ TYPE_SCHEMA

Die DEREF-Funktion ermittelt den strukturierten Typ zu einer OID oder einer Referenzspalte. Das Ergebnis kann dann in den anderen drei Funktionen verwendet werden. TYPE_ID ermittelt die interne ID zu einem Datentyp, TYPE_NAME ergibt den Namen und TYPE_SCHEMA das Schema des Typs.

Beispiel:

```
⇒ SELECT TYPE_NAME(DEREF(OID)), name FROM Personen
```

6.6 DISTINCT Datentypen

Jeder Datenwert, der in der Datenbank gespeichert wird, hat einen spezifischen Datentyp, der die Darstellung des Werts festlegt sowie die darauf anwendbaren Operationen. Beim Aufbau einer Datenbank entscheidet man sich gelegentlich, einen vordefinierten Typ in bestimmter Weise zu verwenden. So werden beispielsweise Altersangaben in Integerwerten gespeichert, geometrische Winkel in DOUBLE-Werten oder Geldbeträge mit Hilfe von DECIMAL(13,2). Man hat dabei möglicherweise bestimmte Regeln über die Berechnungen im Kopf, die in einem solchen Fall Sinn machen. Es macht beispielsweise Sinn, zwei Geldbeträge zu addieren oder zu subtrahieren, nicht aber sie zu multiplizieren. Es macht auch keinen Sinn, eine Altersangabe mit einem Geldbetrag zu vergleichen oder die beiden zu addieren.

SQL stellt dem Benutzer eine Möglichkeit bereit, derartige spezialisierte Verwendungen von Datentypen und die dazugehörigen Regeln zu deklarieren. Das System stellt dann diese Regeln sicher, es dürfen nur die als sinnvoll definierten Operationen durchgeführt werden. Würde man ein Alter mit einem Geldbetrag vergleichen, führt dies zu einer Fehlermeldung. Die Typsicherheit wird damit verstärkt.

Man deklariert eine spezialisierte Verwendung von Daten durch das Erzeugen eines neuen, eigenen Datentyps, eines so genannten Distinct-Typs. Jeder Distinct-Typ hat mit einem der vordefinierten Typen seine interne Darstellung gemeinsam. Letzterer heißt deshalb der Quell- oder Basistyp des Distinct-Typs. Abgesehen von der gemeinsamen Darstellung wird der Distinct-Typ als separater Datentyp betrachtet, der von allen anderen verschieden ist (daher der Name). Ein Exemplar oder eine Instanz eines Distinct-Typs wird nur mit einer anderen Instanz desselben Typs als vergleichbar betrachtet.

```
/-----\
CREATE DISTINCT TYPE std-typname
    AS Quelltypname [FINAL] [WITH COMPARISONS]
\-----/
```

Beispiele:

```
⇒ CREATE DISTINCT TYPE Geld
    AS DECIMAL(13,2) WITH COMPARISONS
⇒ CREATE DISTINCT TYPE Video AS BLOB(100M)
```

Der Zusatz WITH COMPARISONS hat IBM bei DB2 eingeführt, er dient als Erinnerung, dass Instanzen des neuen Distinct-Typs miteinander verglichen werden können. Hierbei stehen die Vergleichsoperatoren $=, \leq, <, >, \geq, \neq$ zur Verfügung. Die Bedeutung des Vergleichsoperators ist dabei dieselbe wie beim jeweiligen Basistyp. Da das System automatisch weiß, wie man Werte eines Distinct-Typs vergleicht, kann man ferner die Sprachelemente ORDER BY, GROUP BY und DISTINCT auf Attribute von einem Distinct-Typ anwenden. Man kann eindeutige und nicht-eindeutige Indizes über ein solches Attribut anlegen.

Folgende Anfragen werden beispielsweise durch Distinct-Typen verhindert:

```
CREATE DISTINCT TYPE USDollar AS REAL;
CREATE DISTINCT TYPE Euro AS REAL;

CREATE TABLE US_SALES
    (CustNo INT, Total USDollar, ...)
CREATE TABLE EURO_SALES
    (CustNo INT, Total Euro, ...)

SELECT E.CustNo
    FROM US_SALES U, EURO_SALES E
    WHERE U.CustNo = E.CustNo AND
        - U.Total > E.Total
        - U.Total > 1000
        - E.Total = 500
```

Für einen Vergleich muss der Vergleichswert vom selben Typ sein oder gecastet werden. Im obigen Select wäre damit eine korrekte WHERE-Bedingung `E.Total > Euro(1000)`.

6.6.1 Casting-Funktionen

Wenn man einen Distinct-Typ erzeugt, werden automatisch zwei Casting-Funktionen zum Konvertieren zwischen dem Distinct-Typ und seinem Basistyp angelegt. Erzeugt man beispielsweise einen Distinct-Typ Alter basierend auf Integer, so erzeugt das System automatisch die Casting-Funktionen `alter(Integer)` und `integer(alter)`.

Sind Alter1 und Alter2 beides Attribute vom Typ Alter, der auf Integer basiert, und ist für den Typ Alter kein Operator '+' definiert, so führt der Ausdruck 'Alter1+Alter2' zu einem Fehler. Der Ausdruck 'Alter(Integer(Alter1) + Integer(ALter2))' ist dagegen korrekt. Dieser Ausdruck ist eine Möglichkeit, dem SQL-Interpreter mitzuteilen, dass ich weiß was ich tue. Die Konvertierungsfunktionen zwischen Distinct-Typen und ihren Basistypen sind sehr effizient, weil Quelltyp und Distinct-Typ dieselbe Darstellung besitzen, so dass keine echte Konvertierung erforderlich ist.

6.6.2 Benutzung von Distinct-Typen

Distinct-Typen sind ein nützliches Werkzeug zum Schutz der Typsicherheit eines Programms. Man kann mit ihnen sicherstellen, dass verschiedene Arten von Daten nicht auf unsinnige Weise kombiniert werden. Insbesondere werden bis auf die Vergleichsoperatoren keine Operatoren vom Basistyp an den Distinct-Typ weitergegeben. Dadurch wird die Multiplikation von Beträgen vermieden, aber auch deren Addition. Nun ist es wahrscheinlich nicht besonders sinnvoll, einen Distinct-Typ zu definieren, dessen einzige Operationen das Konvertieren in den Basistyp und Vergleichen sind. UDB erlaubt es deshalb die Semantik eines Distinct-Typ durch die Definition von Operatoren zu erweitern.

Zum Verständnis des Verhaltens eines Distinct-Typs muss man sich klarmachen, dass das System alle arithmetischen Operatoren wie + und * als Funktionen betrachtet. Der Operator + wird beispielsweise als Funktion mit dem Namen '+' betrachtet, die zwei Integer-Argumente erwartet und eine Integerzahl als Funktionsergebnis hat. Man kann zum Aufruf deshalb auch die funktionale Darstellung verwenden: ''+''(Gehalt, Bonus) Das Funktionssymbol muss dabei in doppelten Hochkommas stehen, so dass das System erkennt, dass es sich um einen Namen handelt.

Die vordefinierten Datentypen sind mit einer Sammlung von vordefinierten Funktionen ausgestattet. Nach dem Erzeugen eines Distinct-Typs kann man auch für diesen Operatoren definieren. Dies geschieht über die Erzeugung neuer Funktionen, so genannter *quellenbasierter Funktionen*, die auf dem Distinct-Typ operieren und die Semantik der jeweiligen Funktion des Basistyps duplizieren. So kann man beispielsweise festlegen, dass der Distinct-Typ Gewicht die arithmetischen Operatoren + und - sowie die Spaltenfunktionen sum und avg von seinem Basistyp erben soll. Eine Multiplikation von Gewichten ist dann nicht möglich.

Man kann auch über ein reines Vererben von Funktionen des Quelltyps hinausgehen und einen Distinct-Typ mit einer eigenen Semantik versehen. Dies geschieht über die Erzeugung *externer Funktionen*, die in einer externen Programmiersprache geschrieben werden und auf dem einzigartigen Typ operieren. Da man diese Funktionen selbst implementiert, können sie frei definierbare Aufgaben durchführen. Benutzt man dabei für die externe Funktion einen Namen wie '+', kann diese Funktion auch in Infix-Schreibweise verwendet werden.

Beispiel:

```
⇒ CREATE DISTINCT TYPE GELD
      AS INTEGER WITH COMPARISONS;
CREATE FUNCTION ''+''(Geld,Geld) RETURNS Geld
      SOURCE sysibm.''+''(Integer, Integer);
CREATE FUNCTION ''-''(Geld,Geld) RETURNS Geld
      SOURCE sysibm.''-''(Integer, Integer);
CREATE FUNCTION ''*''(Geld,Integer) RETURNS Geld
      SOURCE sysibm.''*''(Integer, Integer);
CREATE FUNCTION max(Geld) RETURNS Geld
      SOURCE sysibm.max(Integer);
```

6.7 Objektrelationale Möglichkeiten bei DB2

IBM unterstützt den Referenztypkonstruktor als einzigen unbenannten Typkonstruktor. Die Definition eines Referenztyps entspricht der Notation aus Standard-SQL. Bei IBM muss der refe-

renzierte Typ ein strukturierter Typ sein und der Scope einer Referenz muss wie im Standard auch, auf eine typisierte Tabelle oder Sicht festgelegt werden. In DB2 sind alle Operationen auf die Referenztypen anwendbar, die auch im Standard von SQL verfügbar sind.

Beim Anlegen von Typen mit CREATE TYPE muss immer der Zusatz MODE DB2SQL hinzugefügt werden. Er hat keine Bedeutung und kann in seiner Sinnhaftigkeit durchaus hinterfragt werden, da diese Angabe bei IBM obligatorisch ist und keine alternative Option angegeben wird. Ansonsten ist DB2 beim CREATE TYPE konform zu den in diesem Kapitel dargestellten Stanard.

6.8 Objektrelationale Möglichkeiten bei Oracle

Wie IBM unterstützt auch Oracle die Definition des Referenztypkonstruktors als einzigen unbenannten Datentyp. Die Deklaration eines Referenztyps ist dem Standard sehr ähnlich und weicht nur geringfügig ab:

```
/-----\
|Referenzspaltenname| REF |Objekttypname| SCOPE IS |Objekttabellenname|
\-----/
```

Die Angabe der SCOPE-Klausel ist bei Oracle nicht zwingend nötig. Sie kann angegeben werden, um die Referenzspalte auf eine bestimmte Objekttabelle oder Objektview zu beschränken. Alternativ kann eine derartige Einschränkung bei Oracle auch über eine eigene Tabellenbedingung definiert werden:

```
/-----\
|SCOPE FOR (|Referenzspalte|) IS |Objekttabellenname|
\-----/
```

Im Gegensatz zum Standard ist bei dieser Variante vorteilhaft, dass die SCOPE-Klausel einer Referenztabelle nicht zwingend auf eine einzelne Tabelle beschränkt werden muss, um die Referenzspalte dereferenzieren zu können.

In Oracle ist es möglich Tabellentypen über CREATE TYPE zu definieren. Über einen derartigen Tabellentyp wird eine geordnete und unbegrenzte Multimenge erzeugt. Die Elemente der Tabelle besitzen jeweils denselben Tupeltyp und es sind alle gültigen Datentypen verwendbar. Die benutzerdefinierten Tabellen können bei Oracle geschachtelt werden, dabei ist jedoch zu beachten, dass keine referentiellen Integritätsbedingungen unterstützt werden, wodurch die Inhalte der inneren Tabelle mit den Inhalten der äußeren Tabelle verbunden werden können.

Syntaktisch gibt es bei CREATE TYPE einige weitere Unterschiede: Der Wurzeltyp wird wie folgt angelegt:

```
/-----\
CREATE TYPE |Typname| AS OBJECT ( |Attributdefinitions- und Methodendeklarationsliste| [|Abbildung- oder Ordnungsmethode|] ) [[NOT] FINAL] [[NOT] INSTANTIABLE]
\-----/
```

Ein Subtyp kann, wie im Folgenden dargestellt, erzeugt werden:

```
/-----\
CREATE TYPE |Typname| UNDER |Supertypname| ( |Attributdefinitions- und Methodendeklarationsliste| [|Überschreibende Abbildungsmethode|] ) [[NOT] FINAL]
[[NOT] INSTANTIABLE]
\-----/
```


Kapitel 7

Architektur von Datenbanksystemen

Wie in Kapitel 1 schon vorgestellt, ergeben sich für den Anwender eine Reihe von Grundanforderungen an ein Datenbanksystem:

Zentrale Kontrolle über die operativen Daten

Die Verwaltung der im allgemeinen auf dem Sekundärspeicher angelegten Datenbank und aller damit zusammenhängender Vorgänge obliegt einer zentralen Kontrollinstanz, welche insbesondere in der Lage ist, Veränderungen an konkreten Daten durchzuführen und zu überwachen, so dass Inkonsistenzen vermieden werden können.

Mehrbenutzerbetrieb

Alle Benutzer können zeitgleich auf physischer Ebene mit dem gleichen Datenbestand arbeiten, so dass einheitliche Integritätskontrollen sowie Schutz- und Sicherungsmechanismen anwendbar sind.

Hoher Grad an Datenunabhängigkeit:

Während konventionelle Programme mit Dateizugriff sehr eng mit den Speicherungsstrukturen gekoppelt sind, kann durch eine logische Datenunabhängigkeit die Speicherung der Daten von den Anwendungen getrennt werden, so dass bei Änderungen der Datenstrukturen (z.B. zu Optimierungszwecken) die Anwendungen nicht geändert werden müssen. Dabei ist es auch möglich, einzelnen Benutzern oder Benutzergruppen eine logische Sicht auf die Datenbank zu geben, so dass die Daten anwendungsbezogen strukturiert sind.

Effizienz der Anwendungen:

Ein gutes Leistungsverhalten ist Aufgabe des Datenbanksystems, nicht des Anwendungsprogramms. Das Ziel der Datenunabhängigkeit läuft dabei entgegengesetzt zur Effizienz. Je loser die Bindung zwischen Daten und Programmen ist, desto langsamer ist der Zugriff.

Telefonvermittlungslösungen müssen beispielsweise mehr als 15.000 Benutzerprofile pro Sekunde aus einer Datenbank lesen und jeweils einen Abrechnungssatz schreiben. Management-Informationssysteme müssen Ad-hoc-Anfragen auf 1 TB abarbeiten, wobei ein Durchsatz von 5 Transaktionen pro Sekunde (5 TPS) und ein 'worst-case' von 30 Sekunden Antwortzeit zu erreichen ist. Weitere Gebiete mit hohen Leistungsforderungen sind der Aktienhandel, Reservierungssysteme, etc.

Die Architektur eines Datenbanksystems sollten (neben anderen) diese Ziele unterstützen.

7.1 Die ANSI/SPARC-Schemaebenen

In jedem Datenmodell ist es wesentlich, zwischen der Beschreibung einer Datenbank und der Datenbank selbst zu unterscheiden. Die Beschreibung wird im sogenannten Datenbankschema festgelegt. Dies wird als weitgehend zeitinvariant betrachtet, da sich die Datenbankstruktur nur selten ändert.

Der ANSI/SPARC-Architekturvorschlag basiert im wesentlichen auf der Identifikation von drei unterschiedlichen Abstraktionsebenen und den ihnen zugeordneten Schemata und zielt auf Datenunabhängigkeit ab. Auf jeder der drei Ebenen können jeweils verschiedene Typen von Daten und insbesondere Schemata identifiziert werden. Speziell wird dabei unterschieden zwischen:

- ▶ Physische Datenorganisation
- ▶ Logische Gesamtsicht der Daten
- ▶ Einzelne differenzierbare Benutzersichten

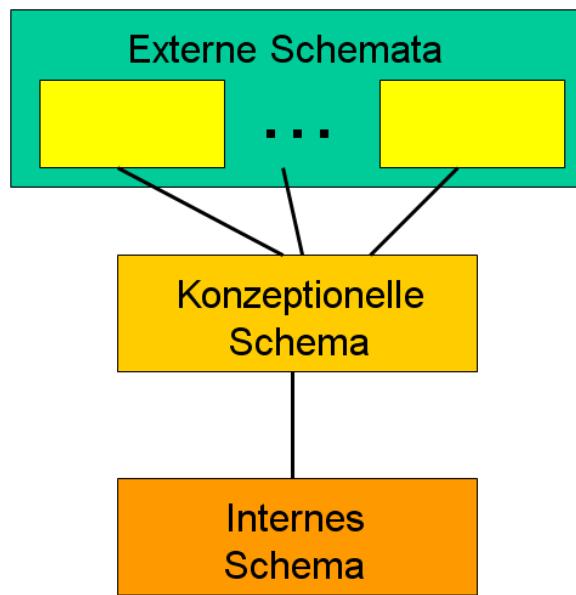


Abbildung 7.1: 3-Ebenen-Datenbankarchitektur nach ANSI/SPARC

Diese Einteilung in drei Abstraktionsebenen ist ein Teilergebnis der Arbeit des Standards Planning and Requirements Committee (SPARC) des American National Standards Institute (ANSI) und wird als ANSI/X3/SPARC-Architekturmödell für Datenbanken bezeichnet.

Die interne Ebene liegt dabei dem physikalischen Speicher am nächsten, wobei die Daten hier als interne Records (Datensätze) betrachtet werden. Diese Sicht der Daten wird im internen Schema festgelegt, in dem Informationen über die Art und den Aufbau der verwendeten Datenstrukturen einschließlich deren Definition und spezieller Zugriffsmechanismen enthalten sind. Auf der internen Ebene wird ferner die Lokalisierung der Daten auf den zur Verfügung stehenden Sekundär-Speichermedien geregelt.

Auf der konzeptionellen Ebene wird die logische Gesamtsicht aller Daten in der Datenbank und ihre Beziehungen untereinander im konzeptionellen Schema repräsentiert. Dazu bedient man sich der sprachlichen Mittel des Datenbankmodells, welches insbesondere von der internen Schicht abstrahiert. Zur Gewährleistung physischer Datenunabhängigkeit ist dabei wesentlich, dass dieses

Schema von Datenstruktur- oder Zugriffsaspekten frei ist. Das konzeptionelle Schema beinhaltet ausschließlich eine Definition des Informationsgehalts der Datenbank.

Die externe Ebene umfasst alle individuellen Sichten (Subschemata) der einzelnen Benutzer oder Benutzergruppen auf der Datenbank. Diese Sichten (Views) werden jeweils in einem externen Schema beschrieben, welches genau den Ausschnitt der konzeptionellen Gesamtsicht enthält, den der Benutzer benötigt. Andere Daten bleiben dem Benutzer verborgen.

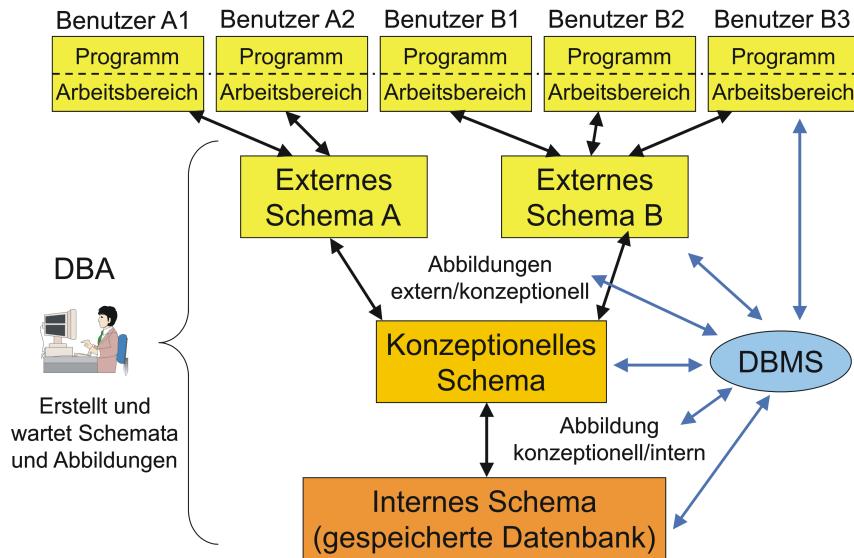


Abbildung 7.2: Schnittstellen im ANSI/SPARC-Vorschlag

In Abbildung 7.2 wird der ANSI/SPARC-Vorschlag noch etwas weiter detailliert. Die Aufgaben des DBMS erstrecken sich von der Verwaltung der gespeicherten Datenbank über die Abbildungen, die durch die verschiedenen Schemata festgelegt werden, bis zu den Arbeitsbereichen der Programme, in denen die gelesenen und die zu schreibenden Datenobjekte abzuholen bzw. abzulegen sind.

7.2 Komponenten eines Datenbanksystems

DBVS sind von ihrem Aufbau und ihrer Einsatzorientierung in hohem Maße generische Systeme. Sie sind so entworfen, dass sie flexibel durch Parameterwahl und ggf. Einbindung spezieller Komponenten für eine vorgegebene Anwendungsumgebung konfigurierbar sind. Hierzu werden die Schemata auf den ANSI/SPARC-Ebenen nach Bedarf erstellt. Die Informationen zum Aufbau des Schemas müssen im Datenbanksystems abgelegt werden. Der Bereich hierfür wird als Data Dictionary oder Systemkatalog bezeichnet, die Beschreibungsinformationen werden auch Metadaten genannt. Schemabeschreibungen werden zur Laufzeit eines Datenbanksystems gelesen, interpretiert und ständig aktualisiert. Da sich diese Metadaten auch aufgrund von Strukturänderungen verändern können, benötigt ein Datenbanksystem eine vollständige Metadatenverwaltung. Beschreibungsdaten werden zur Realisierung der Aufgaben in jeder Schicht benötigt.

Damit ein Datenbanksystem auch nach einem Systemfehler die Wiederherstellung des aktuellsten, konsistenten Datenbestands garantieren kann, werden im laufenden Betrieb Log-Informationen in eine Datei geschrieben. Bei einem Neustart des Systems kann hierdurch ermittelt werden, wann und wie das System zuletzt beendet wurde und welche Änderungen an den Datensätzen nicht dem ACID-Prinzip genügen und damit überarbeitet werden müssen.

Ein Datenbanksystem benötigt damit drei Datenbestände:

- ▶ Die Datenbank
- ▶ Das Data Dictionary
- ▶ Die Loginformationen

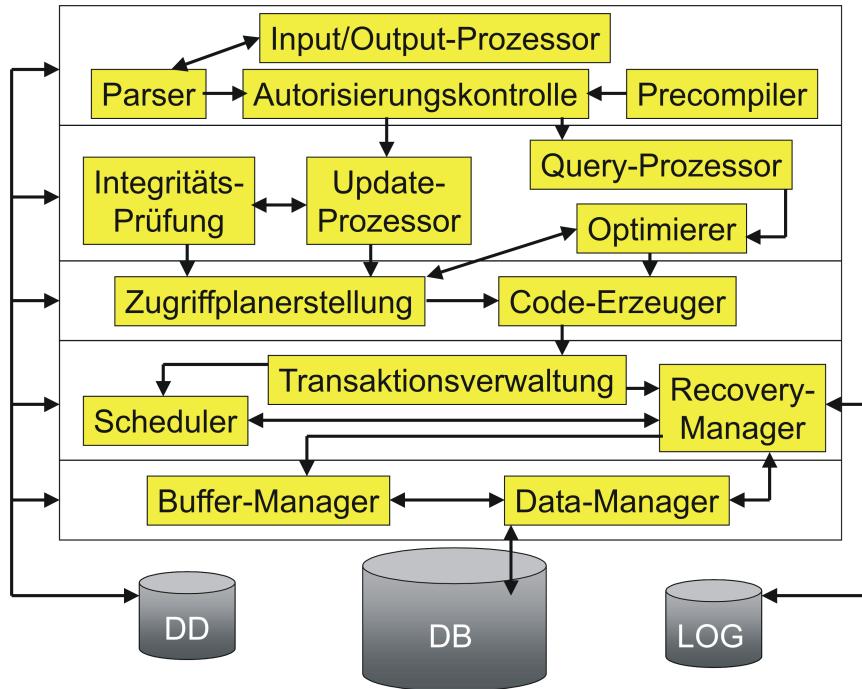


Abbildung 7.3: Komponenten eines DBMS

Abbildung 7.3 zeigt die wichtigsten Komponenten eines DBMS. Dem Benutzer unmittelbar zugeordnet ist ein Input/Output-Prozessor, der Kommandos entgegennimmt und entweder Erfolgs- oder Fehlermeldungen zurückliefert. Bei dieser Komponente handelt es sich technisch häufig um einen Monitor oder Supervisor-Prozess, der in Abhängigkeit vom jeweiligen Benutzerauftrag andere Komponenten aufruft bzw. Prozesse anstößt.

Ein an die Datenbank gerichteter Benutzerauftrag wird an einen Parser übergeben, der zunächst eine syntaktische und semantische Analyse durchführt. Dabei werden Dictionary-Informationen benötigt um beispielsweise zu überprüfen, ob die angesprochene Tabelle existiert. Für Anweisungen, die in einem Programm enthalten sind, übernimmt dies der Precompiler. Für beide Kommandoarten ist eine Autorisierungskontrolle durchzuführen, die beispielsweise feststellt, ob der Benutzer mit den angesprochenen Daten arbeiten darf. Als Resultat dieser ersten Schritte liegt der ursprüngliche Benutzerauftrag in einer internen Form, beispielsweise in einem Ausdruck der Relationalen Algebra, vor.

Bei der weiteren Verarbeitung der Zwischenform wird hier zwischen Änderungen und Selektionen unterschieden. Bei Änderungen muss das Datenbanksystem die festgelegten Integritätsbedingungen bezüglich der Änderung überprüfen. Dies kann bei Abfragen entfallen. Hier wird jedoch ein Optimierer benötigt, der die Abfrage geschickt umformt, so dass eine effiziente Ausführung möglich ist. Dies ist auch bei Änderungen mit WHERE-Bedingung erforderlich, hierbei handelt es sich um eine Kombination von Änderung und Abfrage.

Die nächste Aufgabe des DBMS besteht dann in der Feststellung der auf die benötigten Daten verfügbaren Zugriffstrukturen, der Auswahl eines möglichst effizienten Zugriffspfads und der

Erstellung eines Zugriffs- bzw. Ausführungsprogramms, d.h. einer Code-Generierung für den Benutzerauftrag.

Im Allgemeinen arbeitet ein Benutzer nicht exklusiv mit einer Datenbank, die Operationen müssen mit anderen Benutzern synchronisiert werden. Dazu verfügt das DBMS über einen Transaktionsmanager, der die Datenbank nach außen für jeden einzelnen Benutzer wie ein exklusiv verfügbares Betriebsmittel erscheinen lässt. Der Scheduler sorgt dabei für eine sinnvolle Abarbeitungsreihenfolge der Befehle aller Benutzer. Aus der Sicht des einzelnen Benutzers arbeitet das DBMS immer nach dem Alles-oder-Nichts-Prinzip. Diese wird durch den Recovery-Manager gewährleistet, der Änderung unvollständiger Transaktionen zurücknimmt.

Auf der untersten Ebene ist die Speicherverwaltung des Systems angesiedelt. Die Pufferverwaltung dient dabei der Verwaltung des Hauptspeichers des betreffenden Rechners, der Data Manager verwaltet die Speichermedien.

Kapitel 8

Zugriffspfade

Für die Leistungsfähigkeit eines DBS ist es entscheidend, Sätze über inhaltliche Kriterien (Schlüssel) möglichst effizient auffinden zu können. Es sind deshalb Hilfsstrukturen bereitzustellen, um einen Satz oder eine Menge zusammengehöriger Sätze möglichst direkt zu lokalisieren und damit eine sequentielle Suche zu vermeiden. Diese Zugriffshilfen haben ganz allgemein die Bezeichnung Zugriffsstruktur oder Zugriffspfad, die Datenbankhersteller verwenden den Begriff Index. Sie sollen das Aufsuchen von Sätzen wirksam unterstützen, ohne dabei durch den zusätzlich anfallenden Speicherbedarf und den benötigten Wartungsaufwand das gesamte Leistungsverhalten des DBS zu sehr zu belasten. Dabei lassen sich in erster Linie folgende Arten von Zugriffen unterscheiden:

- ▶ Sequentieller Zugriff auf alle Sätze eines Satztyps (Scan)
- ▶ Sequentieller Zugriff in Sortierreihenfolge eines Attributs
- ▶ Direkter Zugriff über einen eindeutigen Schlüssel (Primärschlüssel)
- ▶ Direkter Zugriff über einen mehrdeutigen Schlüssel (Sekundärschlüssel), wobei eine Satzmenge bereitzustellen ist
- ▶ Direkter Zugriff über zusammengesetzte Schlüssel, mehrdimensionale Wertebereiche und komplexe Suchausdrücke
- ▶ Navigierender Zugriff von einem Satz zu einer dazugehörigen Satzmenge desselben oder eines anderen Satztyps.

Neben den bei der Selektion verwendeten Attributen ist auch die Anfrageart für die Bestimmung des optimalen Zugriffspfads von Bedeutung. Hierbei kann man u.a. folgende Grundarten unterscheiden, die von Zugriffspfaden unterstützt werden können:

- ▶ Exakte Anfrage (exact match queries)
- ▶ Präfix-Match-Anfragen
- ▶ Bereichsanfragen (range queries)
- ▶ Extremwertanfragen
- ▶ Join-Anfragen

Wenn kein geeigneter Zugriffspfad vorhanden ist, müssen alle Zugriffsarten durch einen Scan abgewickelt werden. Bei kleinen Tabellen mit weniger als 100 KB an Daten ist dies häufig ausreichend. In der Regel ist der Scan jedoch eine Notlösung. Er ist nur effizient, wenn Anfragen geringe Selektivität aufweisen, d.h. auf große Treffermengen führen. Bei hoher Selektivität von Anfragen ist ein Scan für die Anfrageauswertung unangemessen, da die Antwortzeit mit dem Umfang der Satzmenge steigt.

Grundsätzlich ist es möglich, die Zugriffsinformationen in die Speicherungsstrukturen der Sätze einzubetten (Primärindex) – etwa durch ihre physische Nachbarschaft oder durch Zeigerverketten – oder sie vollkommen separat zu speichern und durch eine geeignete Adressierungstechnik auf die zugehörigen Sätze zu verweisen (Sekundärindex). Bei ihrer separaten Speicherung kann die Verteilung und Zuordnung der Sätze beliebig sein, während im Falle ihrer Einbettung die physische Position der Sätze durch die Charakteristika des betreffenden Zugriffspfads bestimmt und nicht mehr frei wählbar ist. Deshalb können auch nur ein Primärindex, aber beliebig viele Sekundärindizes angelegt werden.

Generell können zwei Klassen von Zugriffspfaden unterschieden werden:

- ▶ Zugriffspfade für Primärschlüssel, die bei gegebenem eindeutigen Schlüssel auf **den** zugehörigen Satz führen. Anfragen der Art
`SELECT * FROM Pers WHERE PNR = 4711`
werden hierdurch unterstützt. Dies sollte nicht mit dem Begriff Primärindex verwechselt werden.
- ▶ Zugriffspfade für Sekundärschlüssel, mit denen **alle** Sätze eines Satztyps, die in den Werten des Schlüsselattributs übereinstimmen, gewählt werden. Anfragen der Art
`SELECT * FROM Pers WHERE Ort = 'RV'`
werden mit diesen Pfaden unterstützt.

Bei beiden Beispielen handelt es sich um eindimensionale Zugriffspfade, da die Zugriffsstruktur jeweils nur die Suche nach jeweils einem Schlüsselwert unterstützt. Eindimensionale Zugriffspfade lassen sich nach Art des eingesetzten Suchverfahrens durch das in Abbildung 8.1 dargestellte Schema klassifizieren.

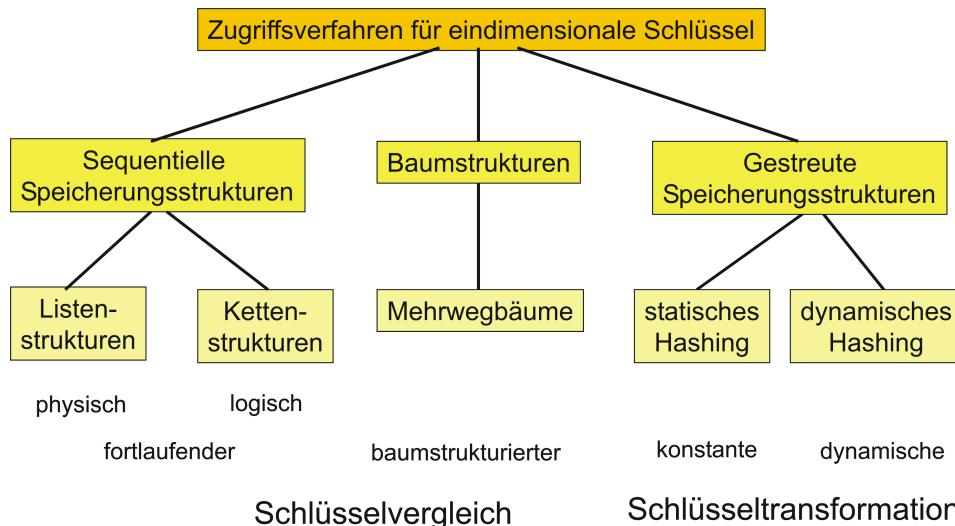


Abbildung 8.1: Klassifikation der eindimensionalen Verfahren

8.1 Primärindizes

Primärindizes werden bevorzugt für den Primärschlüssel eingesetzt, dies ist aber kein Muss. So können andere Attribute für einen Primärindex genutzt werden, auch kann für die Primärschlüsselwerte ein Sekundärindex verwendet werden. Zugriffspfade für Primärschlüssel sind für DBS von besonderer Bedeutung, da bei jedem neu eingefügten Satz überprüft werden muss, ob nicht schon ein Satz mit gleichem Schlüsselwert existiert. Ohne Zugriffstruktur würde der Einfügevorgang ansonsten stark verlangsamt. Daher werden Primärindizes bevorzugt für den Primärschlüssel und Sekundärindizes für Sekundärschlüssel verwendet.

8.1.1 Sequentielle Zugriffspfade

Alle sequentiellen Organisationsformen sind zugeschnitten auf die fortlaufende Verarbeitung der Sätze eines Satztyps und besitzen im allgemeinen große Nachteile beim wahlfreien Zugriff und beim Änderungsdienst.

8.1.1.1 Sequentielle Listen

Bei sequentiellen Listen sind die Sätze physisch benachbart in den Seiten¹ eines zusammenhängenden Seitenbereichs abgelegt. Dies wird auch als Clusterung bezeichnet. Dabei können die Sätze der Listenstruktur nach dem Primärschlüssel sortiert oder völlig ungeordnet sein, da beim Suchen physisch fortlaufende Vergleiche durchgeführt werden. Bei N Sätzen eines Satztyps und b Sätzen pro Seite ergeben sich im Mittel für das Aufsuchen eines Satzes $N/(2^*b)$ Seitenzugriffe.

Einfügen von K7:

K1	K8	K17
K3	K9	K18
K4	K11	...
K6	K12	

Splitting-Technik:

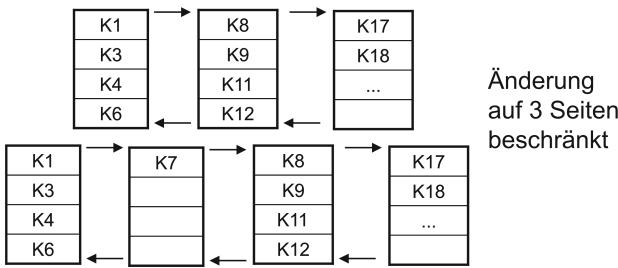


Abbildung 8.2: Einfügen in sequentielle Listen

Im Falle einer Sortierung bleibt das Einfügen eines neuen Satzes nur dann auf ein erträgliches Maß beschränkt, wenn Seiten durch Verkettung beliebig zugeordnet und Verfahren zum Aufteilen und Mischen von Seiten angewendet werden (vgl. Abbildung 8.2). Die Suche nach einem Datensatz über den Primärschlüssel wird bei dieser Variante jedoch nicht beschleunigt, nur der Zugriff in Sortierreihenfolge kann hier sehr schnell erfolgen.

¹DBVSs speichern Datensätze seitenbasiert, d.h. in Blöcken gleicher Größe. Nachfolgende Vergleichsaussagen zu den Zugriffsstrukturen basieren daher auf den Überlegungen, wie viele solcher Blockzugriffe zum Auffinden der Datensätze erforderlich sind. Genaue Informationen zum Aufbau der Blöcke finden sich in Kapitel 11.

8.1.1.2 Gekettete Listen

In geketteten Listen sind alle Sätze eines Satztyps durch Zeiger miteinander verkettet. Das Einfügen von Sätzen wird dadurch erleichtert, dass sie auf einen beliebigen freien Speicherplatz gespeichert werden können. Da die Struktur keinerlei Kontrolle über die Speicherzuordnung ausgeübt wird, erfordert das Aufsuchen eines Satzes durch logisch fortlaufenden Vergleich im Mittel $N/2$ Seitenzugriffe.

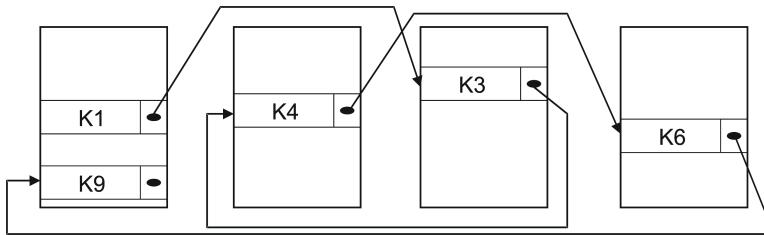


Abbildung 8.3: Kettenstrukturen

Bei den typischen Größenordnungen ergibt sich ganz abgesehen von den Wartungskosten ein so hoher Zugriffsaufwand für die sequentielle Speicherungsstrukturen, dass sie in einer Datenbankumgebung für den Primärschlüsselzugriff nicht in Frage kommen.

8.1.2 Baumstrukturierte Zugriffspfade

Im Rahmen dieser Betrachtungen kann man die Vielzahl der Konzepte und Verfahren weder einführen noch gründlich diskutieren oder gar vergleichen. Hierzu gibt es eine große Anzahl von Fachbüchern im Rahmen des Themas 'Algorithmen und Datenstrukturen'. Nachfolgend werden nur die für den DBS-Einsatz wichtigen Verfahren betrachtet und einige ihrer bedeutenden Eigenschaften vertieft.

8.1.2.1 Binäre Suchbäume

Die weitaus meisten Vorschläge betreffen binäre Suchbäume. Von ihren strukturellen Eigenschaften sind sie nur für Anwendungen gedacht, die vollständig im Hauptspeicher ablaufen, da in ihren Zuordnungsregeln und Suchalgorithmen Seitengrenzen keine Berücksichtigung finden. Binäre Suchbäume lassen sich danach klassifizieren, ob die Zugriffswahrscheinlichkeiten zu ihren Elementen bei Aufbau und Wartung der Struktur eine Rolle spielen oder ob für alle Elemente die gleiche Zugriffswahrscheinlichkeit angenommen wird. Bei den binären Suchbäumen muss vor allem eine in gewissen Schranken ausgewogene Höhe des Baums garantiert werden, damit er auch im ungünstigsten Fall seine Degenerierung zu einer linearen Liste vermieden wird. Die bekanntesten Vertreter sind höhenbalancierte Suchbäume wie der AVL-Baum und seine Erweiterungen sowie gewichtsbalancierte Suchbäume wie der BB[α]-Baum und der WB-Baum. Solange sich diese Konzepte nicht mit geeigneten Regeln der Seitenzuordnung verknüpfen lassen, haben sie für die Implementierung von Zugriffspfaden in Datenbanksystemen keine Bedeutung. Hier wird deshalb auf eine detaillierte Betrachtung verzichtet.

8.1.2.2 Mehrwegbäume

Die für den DB-Einsatz relevanten baumstrukturierten Organisationen sind für Seitenstrukturen konzipiert und unterstützen sowohl den wahlfreien Schlüsselzugriff als auch die sortierte Verarbeitung aller Sätze. Vom Kostenaspekt gesehen bieten sie einen ausgewogenen Kompromiss für diese beiden Verarbeitungsoperationen. Aber auch Bereich-, Extremwert- und Präfix-Match-Anfragen

werden unterstützt. Da sie zusätzlich ein günstiges Verhalten bei Änderungsoperationen aufweisen, lassen sie sich in einem breiten Anwendungsspektrum einsetzen.

8.1.2.2.1 B-Bäume B-Bäume sind immer balanciert, das heißt der Weg von der Wurzel zu einem Blatt ist bei jedem Blatt gleich lang. Die Schlüsselmenge sowie die Einfügereihenfolge sind dabei ohne Bedeutung, das Einfügen neuer Datensätze führt durch dynamische Baumumstrukturierungen immer zu einem balancierten Aufbau.

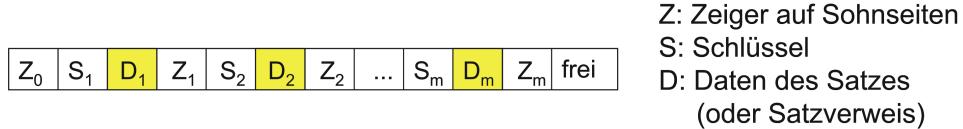


Abbildung 8.4: Format einer Seite im B-Baum

Alle Informationen, die ein Knoten trägt, werden in einer Seite gespeichert (Abbildung 8.4). Dadurch beschreibt die Höhe des Baums beim wahlfreien Zugriff unmittelbar die Anzahl der Seitenzugriffe als relevantes Maß für die Zugriffskosten. Nachfolgend wird der Begriff Knoten als Synonym für Seite aufgefasst. Alle Einträge in der Seite sind nach aufsteigenden Schlüsselwerten sortiert. Für die Zeiger Z_i jeder Seite gilt:

- Z_0 weist auf einen Teilbaum mit Schlüsseln kleiner als S_1 .
- Z_i ($i = 1..m-1$) weist auf einen Teilbaum, dessen Schlüssel zwischen S_i und S_{i+1} liegen.
- Z_m weist auf einen Teilbaum mit Schlüsseln größer als S_m .
- In den Blattknoten sind die Zeiger nicht definiert.

Durch diese Regeln wird gewährleistet, dass alle Schlüssel (und damit Sätze) der Baumstruktur sortiert aufgesucht werden können.

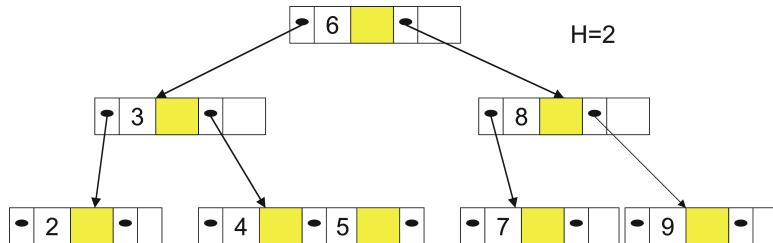


Abbildung 8.5: Ein B-Baum

Ein wichtiges Entwurfsziel für Mehrwegbäume ist es, einen möglichst hohen Verzweigungsgrad zu erzielen um die Höhe so niedrig wie möglich zu halten. Ein typisches Zahlenbeispiel soll den Einfluss der Speicherung der Sätze verdeutlichen. Bei einer Seitengröße von 4 KBytes, Zeiger- und Schlüssellängen von 4 Bytes und einer Satzlänge (Datenteil) von 92 Byte ergibt sich ein Verzweigungsgrad von 40. Enthält der Datenteil aber nur 4 Byte (evtl. durch Auslagern der 92 Byte mit 4 Byte Verweis) kann ein Verzweigungsgrad von über 330 erreicht werden.

8.1.2.2.2 B^* -Bäume Durch B^* -Bäume gelingt es mit einer Einführung von Redundanz im Schlüsselbereich den Verzweigungsgrad von Mehrwegbäumen noch weiter zu erhöhen. Im Gegensatz zum B-Baum, bei dem in allen Einträgen die Information zusammen mit den zugehörigen

Schlüsseln über den ganzen Baum verteilt gespeichert sind, werden beim B^* -Baum die informationstragenden Einträge ausschließlich in die Blattknoten verlagert. Solche Bäume bezeichnet man auch als blattorientierte oder hohle Bäume. Die Schlüssel in den internen Knoten sind Referenzschlüssel und haben ausschließlich Wegweiserfunktion.

Innerer Knoten	$Z_0 S_1 Z_1 S_2 Z_2 S_3 Z_3 \dots S_m Z_m \text{frei}$
----------------	---

Z: Zeiger auf Sohnseite, S:Schlüssel

Blattknoten	$V S_1 D_1 S_2 D_2 \dots S_j D_j \text{frei} N$
-------------	---

D:Daten(verweise), V=Vorgängerzeiger, N=Nachfolgerzeiger

Abbildung 8.6: Seitenformate im B^* -Baum

Die Zeiger P und N in den Blattseiten dienen der Zwei-Weg-Verkettung aller Blattseiten. Da alle Schlüssel sortiert in den Blattseiten gespeichert sind, lässt sich dadurch eine schnelle fortlaufende Verarbeitung aller Sätze in auf- oder absteigender Sortierreihenfolge erreichen. Damit werden exakte Anfragen, Präfix-Match-Anfragen, Bereichsanfragen und Extremwertanfragen unterstützt.

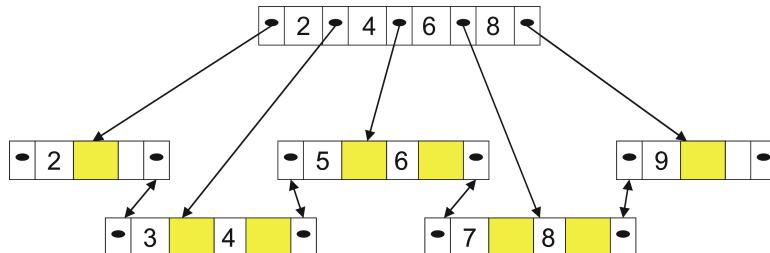


Abbildung 8.7: Ein B^* -Baum

Die redundante Speicherung der Schlüssel führt zu einer Verbreiterung des Baums, da das Format des internen Knotens seinen Verzweigungsgrad bestimmt. So ergibt sich für das Beispiel aus dem Abschnitt zum B-Baum hier ein Verzweigungsgrad von 500.

8.1.2.2.3 Suche in der Seite Neben den Kosten für Seitenzugriffe muss beim Mehrwegbaum der Suchaufwand innerhalb der Seiten als sekundäres Maß berücksichtigt werden. Ein Suchverfahren erfordert eine Folge von Vergleichsoperationen im Hauptspeicher, die bei 500 oder mehr Schlüssel-Verweis-Paare pro Seite durchaus ins Gewicht fallen können. Eine Optimierung der internen Suche erscheint deshalb durchaus gerechtfertigt. Folgende Suchverfahren lassen sich einsetzen:

Systematische Suche: Die Seite wird eintragsweise sequentiell durchlaufen. Bei jedem Schritt wird der betreffende Schlüssel mit dem Suchkriterium verglichen. Unabhängig von einer möglichen Sortierreihenfolge muss im Mittel die Hälfte der Einträge aufgesucht werden.

Sprungsuche: Die geordnete Folge von m Einträgen wird in n Intervalle eingeteilt. In einer ersten Suchphase werden die Einträge jedes Intervalls mit den höchsten Schlüsseln überprüft, um das Intervall mit dem gewünschten Schlüssel zu lokalisieren. Anschließend erfolgt eine systematische Suche im ausgewählten Intervall. Bei dieser Suchstrategie fallen durchschnittlich $n/2+m/(2n)$ Vergleichsschritte an.

Binäre Suche: Die binäre Suche setzt wiederum eine geordnete Folge der Einträge voraus. Bei jedem Suchschritt wird durch Vergleich des mittleren Eintrags entweder der gesuchte Schlüssel gefunden oder der in Frage kommende Bereich halbiert. Die Anzahl der im Mittel benötigten Vergleichsschritte beträgt angenähert $\log_2(m)$.

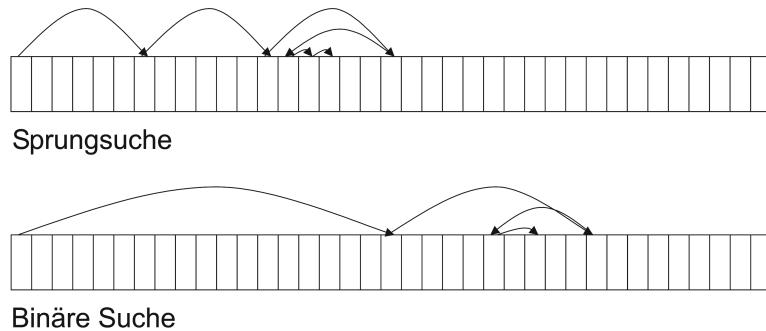


Abbildung 8.8: Suche in einer Seite

8.1.3 Statische Hash-Verfahren

Bei gestreuten Speicherungsstrukturen werden Speicherungs- und Aufsuchoperationen unter der Kontrolle von so genannten Schlüsseltransformations- oder Hash-Verfahren durchgeführt. Der Grundgedanke dieses Verfahrens ist die direkte Berechnung der Speicheradresse eines Satzes aus seinem Schlüssel, ohne (im Idealfall) auf weitere Hilfsstrukturen zurückzugreifen.

8.1.3.1 Hashwertermittlung

Es sei S die Menge aller möglichen Schlüsselwerte eines Satztyps (auch Schlüsselraum genannt) und $A = \{0, 1, \dots, n-1\}$ das Intervall der ganzen Zahlen von 0 bis $n-1$. Eine Hashfunktion $h : S \rightarrow A$ ordnet jedem möglichen Schlüssel $s \in S$ des Satztyps eine ganze Zahl aus A als Adresse zu. Von besonderem Interesse sind hier die Hash-Verfahren für Externspeicher auf Seitenbasis. Dabei wird A als Menge von relativen Seitennummern interpretiert, so dass die berechnete Nummer leicht einer Seite eines zusammenhängenden Dateibereichs zugeordnet werden kann. Eine Seite wird hierbei meist als Bucket (Eimer) bezeichnet.

In der Literatur werden verschiedene Methoden zur Definition der Hash-Funktion, die eine gute Ausnutzung des Speicherbereichs und eine gleichförmige Bucket-Belegung garantieren sollen, vorgeschlagen.

Divisionsrestverfahren: Die Bitdarstellung des Schlüssels k wird als ganze Zahl interpretiert. Durch die Hash-Funktion ' $h(k) = k \bmod q$ ' wird eine ganzzahlige Division durchgeführt. Die Wahl von q bestimmt wesentlich die Gleichverteilung und Speicherausnutzung. Meist wird empfohlen, für q und damit für die Seitenanzahl eine Primzahl zu verwenden.

Faltung: Der Schlüssel k wird in einzelne Bestandteile zerlegt, die als beliebige Partitionen oder überlappende Einheiten gewählt oder durch Verschiebeoperationen erzeugt werden können. Anschließend werden sie additiv, multiplikativ oder durch Boolesche Operationen verknüpft. Das Ergebnis wird als Binärzahl interpretiert. Dies muss dann in geeigneter Weise an den verfügbaren Adressraum angepasst werden.

Multiplikationsverfahren: Der Schlüssel wird mit sich selbst oder mit einer Konstanten multipliziert. Zur Anpassung an eine zulässige Adresse werden bestimmte Bitpositionen aus

dem Ergebnis ausgeblendet. Aus Gründen der Gleichverteilung werden meist Anfangs- und Endbits ausgeblendet.

Basistransformation: Der Schlüssel k wird als Ziffernfolge einer anderen Basis p dargestellt. Zur Bestimmung einer zulässigen relativen Adresse können wiederum Faltung oder Divisionsrestverfahren angewendet werden.

Zufallsmethode: Zur Erzeugung der Hash-Adressen wird ein Pseudozufallszahlengenerator verwendet. Der Schlüssel k dient dabei jeweils als Saat für eine Zufallszahl, aus der die Hash-Adresse – unter Anpassung an den verfügbaren Hashbereich – gewonnen wird.

Ziffernanalyse: Diese Methode setzt voraus, dass die Menge K der zu speichernden Schlüssel bekannt ist. Für jede der m Stellen der Schlüssel k_i wird die Verteilung der Werte in K ermittelt. Die Stellen mit der größten Verteilungsschiefe werden bei der Adressierung nicht berücksichtigt. Aus diese Weise können die Stellen herausgefunden werden, welche die beste Gleichverteilung im vorgegebenen Hash-Bereich garantieren. Die Ziffernanalyse hängt als einziges Verfahren von der spezifischen Schlüsselmenge ab. Sie ist sehr aufwändig, gewährleistet aber gute Gleichverteilungseigenschaften. In DBS lässt sie sich jedoch nicht als allgemeine Methode verwenden, da in praktischen Einsatzfällen die genaue Kenntnis der Schlüsselmenge K nicht vorausgesetzt werden kann.

8.1.3.2 Überlaufbehandlung

Übersteigt die Anzahl der einer Seite zugewiesenen Sätze das Fassungsvermögen eines Buckets, muss eine Kollisionsauflösung (Überlaufbehandlung) durchgeführt werden. In DBS eignen sich hierfür separate Überlaufbereiche – durch Verkettung zusätzlicher Buckets – besonders gut. Bei der Speicherung von weniger als Bucketanzahl * Bucketkapazität Sätzen lässt sich typischerweise ein Zugriffsfaktur von 1.1 – 1.4 erzielen. Selbst wenn die Kapazität des Primärbereichs um 100 Prozent überschritten wird, bleibt der Zugriffsfaktor mit < 2 stabil.

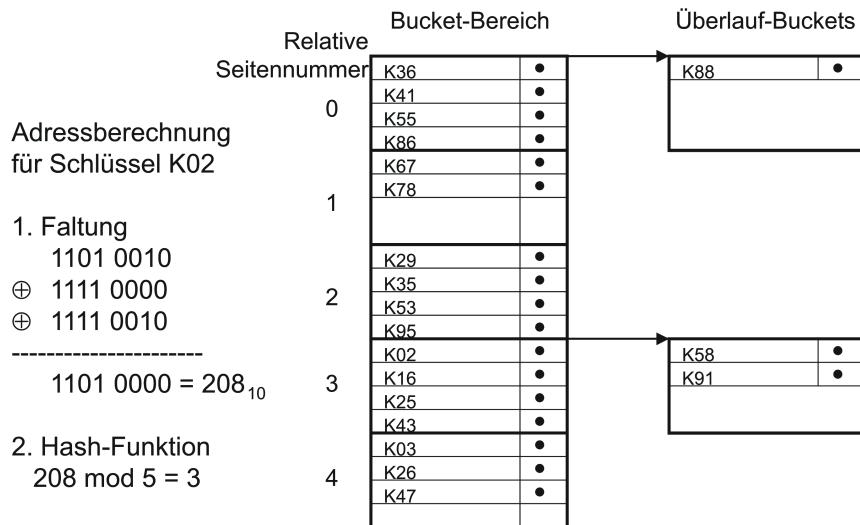


Abbildung 8.9: Überlaufbereiche beim statischen Hashing

8.1.4 Dynamische Hash-Verfahren

Die Notwendigkeit der statischen Zuweisung des Hash-Bereichs bringt bei stark wachsenden Datenbeständen gravierende Nachteile mit sich. Um ein schnelles Überlaufen zu vermeiden, muss er

von Anfang an genügend groß dimensioniert werden, so dass zunächst eine schlechte Speicherausnutzung erzielt wird. Wenn das geplante Fassungsvermögen des Hash-Bereichs überschritten wird, ist die Speicherungsstruktur entweder nicht mehr aufnahmefähig oder es findet zunehmend eine Verdrängung der neuen Schlüssel in den separaten Überlaufbereich statt, so dass der Zugriffsfaktor stetig anwächst. Die sich verschlechternden Zugriffszeiten erzwingen auf Dauer eine Reorganisation derart, dass ein größerer Hash-Bereich anzulegen und alle Schlüssel (Sätze) durch vollständiges Rehashing neu zu laden sind.

In DBS ist dieser Zwang zur statischen Reorganisation mit völliger Neuverteilung der Adressen nicht nur wegen des hohen Zeitbedarfs, der eine längere Betriebsunterbrechung erfordert, sondern auch wegen der vielen Adressverweise aus anderen Zugriffspfaden, die eine isolierte Reorganisation meist verhindern, sehr störend. Wünschenswert ist der Einsatz eines dynamischen Hash-Verfahrens, das

- ▶ ein Wachsen und Schrumpfen des Hash-Bereichs erlaubt,
- ▶ Überlauftechniken und damit eine statische Reorganisation mit vollständigen Rehashing vermeidet,
- ▶ eine hohe Belegung des eingesetzten Speicherplatzes unabhängig vom Wachstum der Schlüsselmenge garantiert und
- ▶ für das Auffinden eines Satzes mit gegebenen Schlüssel nicht mehr als zwei Seitenzugriffe benötigt.

Von den verschiedenen Verfahren, die im Hinblick auf diese Zielsetzung entwickelt wurden, soll hier das erweiterbare Hashing, ein Verfahren mit Indexnutzung, vorgestellt werden. Ein Index (Directory) dient dabei zur Aufteilung der Buckets im Falle eines Überlaufs.

Beim erweiterbaren Hashing enthält eine Split/Merge-Technik zur dynamischen Anpassung des Hash-Bereichs. Auch hier wird zuerst mit Hilfe einer Hashfunktion aus einem Schlüssel k eine Bitfolge k' erzeugt, die zur Adressierung verwendet wird. Dabei werden nicht alle Bits zur Adressierung herangezogen, sondern nur die ersten d , wobei d als globale Tiefe bezeichnet wird. Für jede mögliche Bitkombination der Länge d gibt es einen Eintrag im Directory, jeder Eintrag enthält wiederum einen Zeiger auf einen Bucket, in dem die Datensätze enthalten sind, die zu den vorderen d Bit passen.

Damit nicht für jeden Eintrag im Directory auch ein Bucket angelegt werden muss, können verschiedene Einträge auf den selben Bucket verweisen. Dabei müssen jedoch immer die vorderen d' ($d' \leq d$) Bit des Eintrags übereinstimmen, so dass dann in dem referenzierten Bucket alle Einträge enthalten sind, bei denen die Hashfunktion eine Bitfolge ergibt, die bezüglich der ersten d' Bit mit der Bucketadressierung übereinstimmt. d' wird deshalb auch als die lokale Tiefe des Buckets bezeichnet.

In Abbildung 8.10 ist die Wirkungsweise des erweiterbaren Hashing verdeutlicht. Die Kapazität b der Buckets sei dabei 4. Alle Schlüssel, deren Hashwert in den ausgewählten $d=3$ Bit übereinstimmen, werden über denselben Eintrag des Adressverzeichnisses erreicht. Stimmen bei einem Bucket d und d' überein, so sind nur Schlüssel gespeichert, deren Hashwert die gleiche Bitfolge in den ausgewählten d Positionen besitzen. Im Adressverzeichnis gibt es für diesen Fall genau einen Eintrag für das Bucket. Falls $d' < d$ ist, enthält das Bucket alle Schlüssel, mit einer Übereinstimmung der Hashwerte in der Länge d' , d.h. es verweisen mehrere Einträge des Adressverzeichnisses auf das Bucket.

Solange kein Bucket-Überlauf droht, verlangen Einfügeoperationen keine Sondermaßnahmen. Erst bei einem Einfügeversuch in ein volles Bucket wird eine Strukturänderung des Hash-Bereichs erzwungen. Bei $d' < d$ wird ein Split-Vorgang ausgelöst, der zum Anlegen eines neuen Bucket mit einer wertabhängigen Aufteilung der Schlüsselmenge gemäß den $d'+1$ Bit des Hashwerts und einer

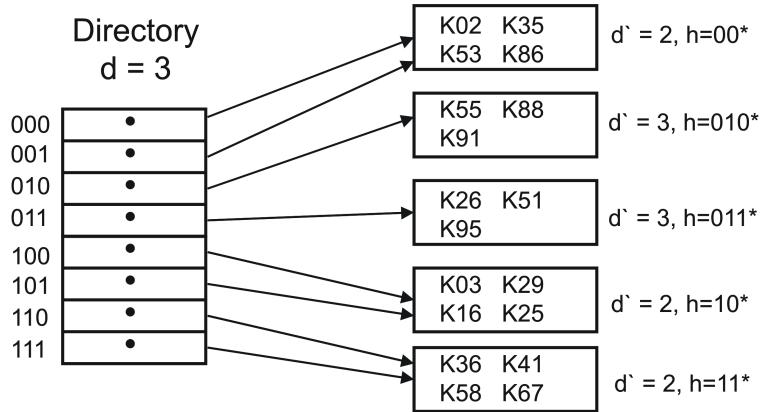
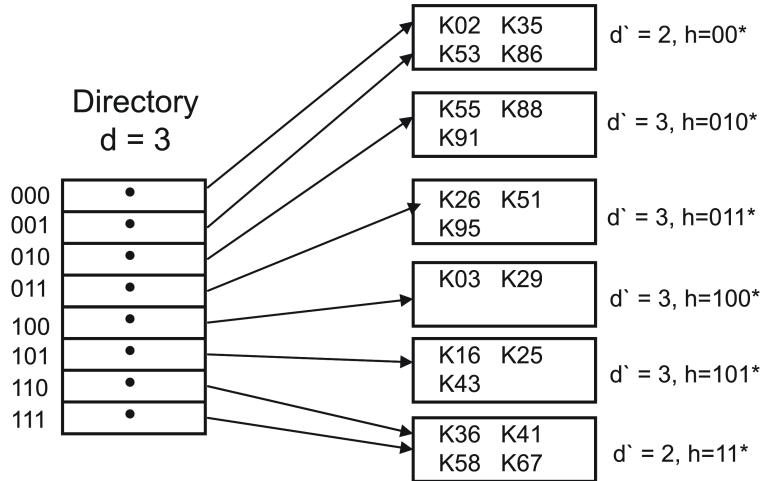


Abbildung 8.10: Erweiterbares Hashing mit Directory

Abbildung 8.11: Überlauf bei $d' < d$

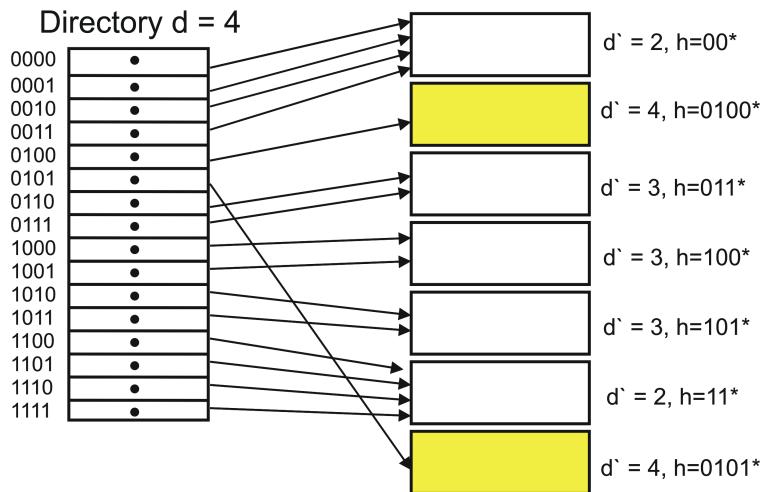
Adaption der betroffenen Verweise im Adressverzeichnis führt. Abbildung 8.11 zeigt die Situation nach Einfügen des Satzes K43 mit Hashwert 101.....

Falls $d = d'$ gilt, ist der Einfügevorgang bei einem vollen Bucket schwieriger. Um das volle Bucket aufzuteilen zu können, muss die lokale Tiefe um 1 erhöht werden, um eine wertabhängige Verteilung der Schlüsselmenge zu erreichen. Ebenso ist sicherzustellen, dass das neu hinzugekommene Bucket adressiert werden kann. Dazu erfolgt auch eine globale Erhöhung der Tiefe um 1, was einer Verdopplung der Einträge des Adressverzeichnisses entspricht (siehe Abbildung 8.12 nach einem Überlauf bei 010^*).

Das erweiterbare Hashing liegt in seiner Zugriffsgeschwindigkeit zwischen statischem Hash-Verfahren und Mehrwegbäumen, da immer ein Zugriffsfaktor von 2 garantiert ist. Da es die Schlüsselsortierung nicht bewahrt, bleibt es vom Funktionsumfang gesehen den Mehrwegbäumen unterlegen.

8.2 Sekundärindizes

Bei Sekundärindizes werden die Datensätze außerhalb der Zugriffsstruktur über Verweise (Tuple Identifier, TIDs) gespeichert. Wie anfangs dargestellt, wird diese Indizierungsform bei Sekundärattributen bevorzugt verwendet. Während bei den Zugriffspfaden für Primärschlüssel jede Suche auf

Abbildung 8.12: Überlauf bei $d' = d$

höchstens einen Satz führt, haben Zugriffspfade für Sekundärschlüssel jeweils mehrere Sätze so zu verknüpfen, dass sie als Ergebnis eines Suchvorgangs bereitgestellt werden können. Insbesondere sollte es möglich sein, verschiedene Suchkriterien in einer Anfrage mit mehreren Zugriffspfaden zu verknüpfen.

Die Invertierungstechnik ist zugeschnitten auf die Auswertung von Anfragen mit mehreren Suchkriterien. In der Zugriffspfadstruktur werden dabei nur die TIDs der Datensätze gespeichert. Anfrageverknüpfungen lassen sich auf mengenalgebraische Operationen auf den TID-Mengen sehr effizient ausführen.

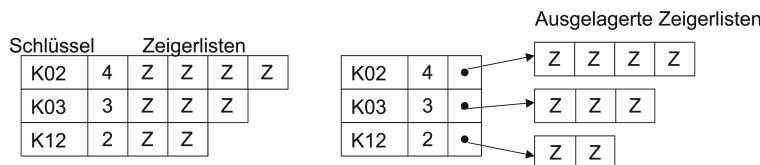


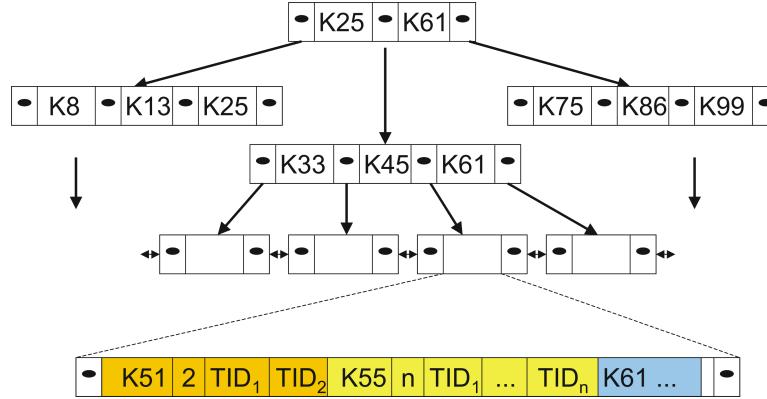
Abbildung 8.13: Verweislisten bei der Invertierung

Die konkrete Implementierung der Invertierung der Sätze eines Satztyps für ein Attribut richtet sich nach den Schemata in Abbildung 8.13. Sowohl B^* -Bäume als auch Hash-Verfahren können mit der Invertierungstechnik eingesetzt werden. Die zu einem Sekundärschlüssel gehörende Verweisliste zeigt auf alle Sätze, die diesen Schlüssel als Attributwert besitzen.

8.3 Bewertung eindimensionaler Zugriffspfade

Als Bewertungskriterien werden hier die drei wichtigsten Verarbeitungsprimitive in einer Zugriffspfadstruktur herangezogen:

- ▶ direkter Zugriff auf einen Satz bei gegebenem Schlüssel
- ▶ sequentielle (sortierte) Verarbeitung aller Sätze
- ▶ Änderungsdienst beim Einfügen oder Löschen eines Datensatzes

Abbildung 8.14: B^* -Baum mit Verweislisten

Zugriffsverfahren	Speicherungsstruktur	Direkter Zugriff	Sequentielle Verarbeitung	Änderungen (ohne Suche)
Fortlaufender Schlüsselvergleich	Seq. Liste gekettete Liste	$O(N)$ $O(N)$	$O(N)$ $O(N)$	≤ 2 ≤ 3
Baumstrukturierter Schlüsselver.	Binärbaum Mehrwegbaum	$O(\log_2 N)$ $O(\log_k N)$	$O(N)$ $O(N)$	2 2
Konstante Schlüsseltransformation	Statisches Hashing mit Überlauf	$1.1 - 1.4$	$O(N \log_2 N)$ mit Sortierung	ca. 1.1
Variable Schlüsseltransformation	Erweiterbares Hashing	2	$O(N \log_2 N)$ mit Sortierung	ca. 1.1

Tabelle 8.1: Vergleich der Zugriffsverfahren

Tabelle 8.1 soll den Aufwand in Bezug auf die Anzahl der Datensätze N zeigen. Die O -Notation zeigt nur die Größenordnung der Kostensteigerung an und verbirgt im Einzelfall konstante Faktoren, deren Größe für die praktische Einsatztauglichkeit einer Struktur von entscheidender Bedeutung sein kann. $O(N)$ bedeutet also, dass die Kosten für die entsprechende Operation linear durch N bestimmt werden. Tabelle 8.2 gibt einen Überblick über die wichtige Systeme und die unterstützten Strukturen.

8.4 Tabellenübergreifende Zugriffspfade

Eindimensionale Zugriffspfade gestatten den direkten Schlüsselzugriff auf alle Sätze eines Satztyps und lokalisieren dabei einen Satz oder eine Menge von Sätzen mit Hilfe eines Primär- oder Sekundärschlüssels. Sie stellen gewissermaßen Basiszugriffsverfahren dar, mit denen sich auch höhere DB-Operationen flexibel und effizient abwickeln lassen. Fallweise ist es jedoch zu empfehlen zur Unterstützung wichtiger Beziehungstypen oder häufig vorkommender DB-Operationen zugeschnittene Zugriffsverfahren zu entwickeln, die den Zugriff über mehrere Satztypen – also einen typübergreifenden Zugriff – beschleunigen, um ein verbessertes Leistungsverhalten zu erzielen.

Mit dem zunehmenden Spezialisierungsgrad der Zugriffsstrukturen nimmt die Einsatzbreite und

	Ingres	Oracle	DB2	Informix
seqquentiell	+	+	+	+
Cluster	-	+	+	-
B-Baum	-	-	-	-
B^* -Baum	+	+	+	+
Hash	+	+	-	-

Tabelle 8.2: Verfahrenseinsatz

damit ihre Nützlichkeit für das Leistungsverhalten des DBS ab. Beim Einsatz solcher Strukturen ist also vermehrt anwendungsbezogenes Wissen und entsprechende Vorplanung Voraussetzung. Nur wenn die DB-Operation passt, sind im Vergleich zu den Basiszugriffsverfahren große Leistungsgewinne zu erzielen. Im folgenden wird als Beispiel eine Zugriffsstruktur, der Join-Index, für den Join als wichtiger relationaler Operator vorgestellt.

Auf relationalen DBS basierende Anwendungen suchen häufig 'von außen kommend' mit Hilfe von Primär- und Fremdschlüssel Tupel in den entsprechenden eindimensionalen Zugriffspfaden. Für einen Join wäre es vorteilhaft, wenn die hierfür erforderliche Zugriffsunterstützung kombiniert in einer physischen Zugriffspfadstruktur verfügbar ist. Da Primär- und Fremdschlüssel auf dem selben Wertebereich definiert sind, ist dies auch gut durchführbar.

Zur Umsetzung dieser Idee kann wieder der B^* -Baum genutzt werden. Sein Indexteil, der nur Wegweiser beherbergt, bleibt bei unverändertem Aufbau von Wurzel- und Zwischenknoten erhalten. Lediglich die Blattknoten erhalten eine andere Organisationsform. Ein Eintrag setzt sich aus einem Schlüsselwert, dem TID als Verweis auf die Vater-Relation, der TID-Liste mit den Verweisen auf die Sohn-Relation und der zugehörigen Längeninformation zusammen. Dieses Konzept lässt sich noch weiter verallgemeinern indem weitere Tabellen mit Attributen auf dem selben Wertebereich mit Hilfe eines einzigen B^* -Baums indiziert werden. Dazu ist das Format der Blattknoten so zu modifizieren, dass mehrere variabel lange Verweislisten für einen Schlüsselwert aufgenommen werden (vgl. Abbildung 8.15).

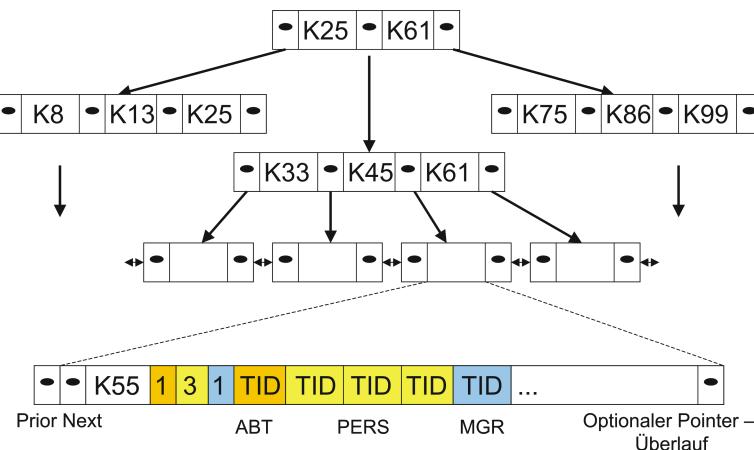


Abbildung 8.15: Beispiel eines Join-Index

In relationalen Datenbanksystemen lässt sich der Nutzen eines solchen Zugriffspfads sehr einfach verdeutlichen. Die folgende SQL-Anfrage bestimmt die Manager von Abteilungen, die mehr als 5 Mitarbeiter haben.

```
SELECT AL.PNR, AL.NAME, A.ANAME
FROM AL, ABT A, PERS P
```

```

WHERE AL.ANR = A.ANR
  AND A.ANR = P.ANR
  AND 5 < (SELECT COUNT(*) FROM PERS Q
             WHERE Q.ANR = A.ANR)

```

Da hier beide Verbunde über ANR spezifiziert sind, können sie direkt auf den verallgemeinerten Zugriffspfad nach Abbildung 8.15 abgewickelt werden. Ein guter Optimierer erkennt auch, dass die COUNT-Bedingung sich unmittelbar aus der Längeninformation ermitteln lässt.

Die verallgemeinerte Zugriffspfadstruktur weist folgende Charakteristika auf:

- ▶ Die Höhe des B^* -Baums ist wegen des hohen Verzweigungsgrads nur geringfügig größer als bei der einfachen Indexstruktur, obwohl in den Blättern wesentlich mehr Zugriffspfadinformationen untergebracht sind. Der direkte Zugriff wird folglich nur unwesentlich langsamer.
- ▶ Da alle Schlüssel im B^* -Baum nur einmal gespeichert werden, resultiert daraus eine Einsparung an Speicherplatz.
- ▶ Der sequentielle Zugriff zu allen Sätzen über einen Index wird langsamer, da eine größere Anzahl an Blattknoten aufzusuchen ist. Gemessen an der gesamten Anzahl der Seitenzugriffe ist dieser Anstieg jedoch minimal.
- ▶ Sie unterstützt in natürlicher Weise die Verbundoperation.
- ▶ Sie bietet große Vorteile bei der Auswertung bestimmter statistischer Anfragen und bei der Überprüfung der referentiellen Integrität und anderer semantischer Integritätsbedingungen.

8.5 Mehrdimensionale Zugriffspfade

Die zuvor diskutierten Zugriffspfadstrukturen auf eine Tabelle sind eindimensional in dem Sinne, dass als Suchschlüssel nur der Wert genau eines Attributs verwendet werden kann. Suchschlüssel oder -ausdrücke, die als Kombination von Werten verschiedener Attribute aufgebaut sind, lassen sich nicht direkt über einen Zugriffspfad auswerten. Es müssen vielmehr mehrere unabhängige Suchvorgänge abgewickelt werden. Dabei wird gewissermaßen die mehrdimensionale Suche mit Hilfe mehrerer eindimensionaler Suchvorgänge simuliert.

Mehrdimensionale Anfragen sind in herkömmlichen Anwendungen durchaus häufiger zu finden. Bei der Verwendung von geografischen und geometrischen Daten, beim CAD- und VLSI-Entwurf sowie beim Information-Retrieval ist ihre effiziente Behandlung noch wichtiger. Außerdem fordert die DBS-Integration neuer Datentypen wie Multimedia-Typen, chemische Formeln oder XML eine angemessene Unterstützung.

8.5.1 Mehrattributzugriff über eindimensionale Pfade

Bevor neue und zusätzliche Zugriffspfadtypen in einem DBVS implementiert und bereitgestellt werden, ist zu prüfen, wie gut die vorhandenen Hilfsmittel die gestellte Aufgabe lösen. Die einfachste Möglichkeit ist das Anlegen von einem eigenen Index für jedes Schlüsselattribut. Abbildung 8.16 ist skizziert, wie auf diese Weise ein zweidimensionaler Schlüsselraum nach den Werten der beiden Schlüsselattribute Key_1 und Key_2 zerlegt werden kann.

Der Zugriff bzgl. einer Suchbedingung ($key1 = k1$) AND ($key2 = k2$) hat so zu erfolgen, dass nacheinander auf beiden Indizes die entsprechende Bedingung überprüft wird. Bei erfolgreicher Suche werden die TID-Listen ermittelt und anschließend der Durchschnitt aus beiden Listen ermittelt. Dieser Durchschnitt entspricht dann der Ergebnismenge. Nachteil dieser Lösung ist jedoch,

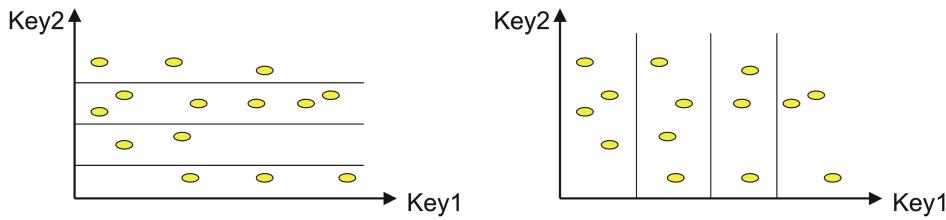


Abbildung 8.16: Getrennte Partitionierung des Schlüsselraums

dass Einfüge- und Löschoperationen verlangsamt werden, da jeweils mehrere Zugriffsstrukturen angepasst werden müssen.

Alternativ kann man die Schlüssel konkatenieren und die Verbindung $Key_1 | Key_2$ als einen Gesamtschlüssel auffassen. Die in dem so aufgebauten B^* -Baum aufgebauten TID-Listen verweisen jeweils auf die Sätze, die genau die betreffende Wertekombination für die Attributgruppe als Inhalt haben. Das Auswahlvermögen eines Mehrattribut-Index entspricht damit der AND-Verknüpfung aller entsprechenden einfachen Indexstrukturen. Die Aufsuchoperation ($key1 = k1$) AND ($key2 = k2$) wird umgesetzt auf die Suchbedingung $Key_1 | Key_2 = k1 | k2$. Exakte Anfragen sowie Einfügen und Löschen einzelner TIDs sind hier unproblematisch.

Jedoch stellen sich Probleme bei allgemeiner Nutzung einer solchen Indexstruktur mit konkatenierten Attributen ein. Zwar kann eine Anfrage der Art ($key1 = k1$) noch akzeptabel unterstützt werden. Anfragen wie ($key2 = k2$) oder ($key1 = k1$) OR ($key2 = k2$) können nicht sinnvoll ausgewertet werden.

8.5.2 Bitmap-Indizes

Bei Bitmap-Indizes handelt es sich formal um eindimensionale Zugriffspfade. Die Vorteile der Struktur werden jedoch beim mehrdimensionalen Zugriff deutlich, so dass diese Struktur erst hier vorgestellt wird. Es sei aber schon im voraus angemerkt, dass sich diese Zugriffsstruktur nur für ganz bestimmte mehrdimensionale Anfragearten eignet, dort aber sehr gute Ergebnisse liefert.

Das Einsatzgebiet von Bitmap-Indizes sind Attribute geringer Selektivität, d.h. Anfragen bzgl. dieser Attribute führen auf große Treffermengen. Beispiele für derartige Attribute sind Ja/Nein-Felder, Farbe oder Jahr. Der Aufbau eines B^* -Baums hätte eine Baumstruktur zur Folge, bei der die Wurzel gleichzeitig Blattknoten ist und bei den wenigen möglichen Werten stehen sehr lange TID-Listen. Diese sehr langen TID-Listen für die wenigen verschiedenen Werte lassen sich auch durch Bitlisten darstellen.

Im Gegensatz zu TID-Listen, die explizite Verweise enthalten, besteht bei Bitlisten der Grundgedanke darin, die zu indexierenden Sätze eindeutig den Bitpositionen einer linearen Bitliste (Bitmap) zuzuordnen. Für jeden Sekundärschlüsselwert wird eine solche Bitliste angelegt, in der die Positionen markiert sind (1-Bit), deren zugeordneter Satz den Wert des Sekundärschlüssels besitzt. Ein Beispiel:

Blau:	000110001000110000110000000001001000
Rot:	1100000000001001000001000000110000001
Weiß:	001000000110000001000001110000000110
Grün:	000001110000000110000110001000110000

Die Länge aller Bitlisten ist für jeden Sekundärschlüsselwert gleich und entspricht der Anzahl an Datensätzen in der Tabelle. Für Anfragen, die den Wert für ein Attribut vorgeben, muss in der Bitliste entsprechen nach den 1er gesucht werden. Der große Vorteil wird aber erst dann deutlich,

wenn die Anfrage mehrere Attribute über einen Suchwert bestimmt und alle Attribute über eine Bitliste indiziert sind. Dann können nämlich die Bitlisten aus den einzelnen Bitlisten kombiniert werden. Sind in der Anfrage die Einzelbedingungen Und-verknüpft, werden die Bitlisten einfach mit AND verknüpft, das Ergebnis enthält die gewünschten Sätze. Gleiches ist mit OR und NOT möglich. Auch das Zählen von Sätzen (Count) kann effizient durchgeführt werden. Zusätzlich benötigen Bitlisten wenig Speicherbedarf, da sie sich aufgrund der Dominanz der 0er sehr gut komprimieren lassen.

Diese seit langem bekannte Bitlistentechnik wird aufgrund von Data-Warehouse-Anwendungen und OLAP (Online Analytical Processing) immer populärer. Bei diesen Anwendungen sind die Daten nach mehreren Dimensionen (Attributen) auszuwerten, wobei häufig mengenalgebraische Verknüpfungen sowie Gruppierungs- und Aggregatfunktionen anfallen. Da in den einzelnen Dimensionen die Anzahl der verschiedenen Werte in der Regel begrenzt ist (beispielsweise Dimensionen wie Farbe, Region, Verantwortlicher), kommen genau die starken solcher Bitlisten zum Tragen.

8.5.3 Grid-Files

Nächster-Nachbar-Anfragen sollen auch dann noch ein Ergebnis liefern, wenn die Auswertung des Suchprädikats auf keine Treffer führt. In einem solchen Fall möchte man das Suchkriterium etwas lockern mit dem Ziel, ein oder mehrere Sätze zu finden, die der vorgegebenen Suchbedingung möglichst nahe kommen. Was 'möglichst ähnlich' heißt, kann dabei nur die Anwendung entscheiden. Deshalb muss diese eine Distanzfunktion vorgeben.

Um diesen Anfragetyp in effizienter Weise bearbeiten zu können, sind geeignete Zugriffspfade und Speicherungsstrukturen anzulegen. Bei der Abbildung sind dabei folgende Eigenschaften zu berücksichtigen:

Erhalten der topologischen Struktur: Eine Kardinalforderung betrifft die Erhaltung der topologischen Struktur der Objekte bei der Speicherzuordnung. Benachbarte Objekte sollen als benachbarte Sätze gespeichert werden. Dies ist notwendig, da Zugriffsanforderungen häufig topologische Bezüge besitzen.

Stark variierende Objektdichte: Objekte sind meist sehr ungleichmäßig verteilt. Deshalb ist es für eine Zugriffsstruktur nicht möglich, die möglichen Werte in ein regelmäßiges Raster aufzuteilen und jedem Rasterelement einem Bucket zuzuordnen. Dies würde eine sehr schlechte Speicherplatzausnutzung ergeben.

Dynamische Reorganisation: Einfüge- und Löschoperationen müssen von einer dynamischen Reorganisation unterstützt werden, wobei sowohl die topologische Struktur als auch eine vernünftige Speicherbelegung erhalten bleiben.

Ein Grid-File ist eine k-dimensionale Zugriffsstruktur, die eine Gleichbehandlung aller k Schlüssel eines Datensatzes sowie den symmetrischen und gleichförmigen Zugriff über diese bietet. Es wurde mit dem Ziel entworfen, ohne Unterschiede in der Vorgehensweise sowohl exakte Anfragen und Anfragen über Teilschlüssel als auch alle Varianten von Bereichsanfragen sowie sogar Nächster-Nachbar-Anfragen zu unterstützen.

Ausgangspunkt ist die Betrachtung des Datenraums. Dieser wird dynamisch durch ein orthogonales Raster (grid) partitioniert, so dass k-dimensionale Zellen (Grid-Blöcke) entstehen. Die in den Zellen enthaltenen Objekte werden Buckets zugeordnet. Benötigt wird dann eine eindeutige Abbildung der Zellen zu den Buckets. Dies wird über das Grid Directory realisiert. Für jede Zelle des Datenraums gibt es im Directory einen Verweis auf einen Bucket. Mehrere Grid-Zellen, die auf den gleichen Bucket verweisen, bilden eine Grid-Region.

Das Prinzip des Datenzugriffs ist damit zweistufig aufgebaut (Abbildung 8.17). Zuerst ermittelt man im Grid-Directory die Buckets, die die gesuchten Sätze enthalten, und danach wird auf die

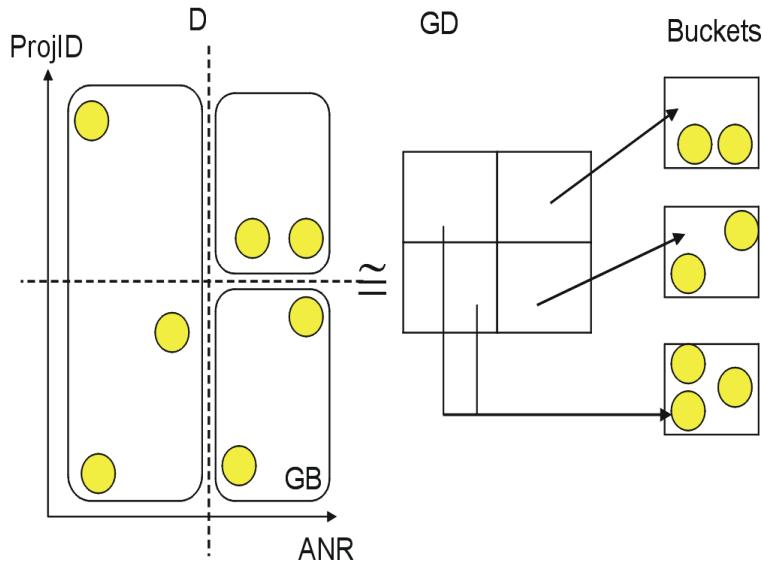


Abbildung 8.17: Zweistufiger Datenzugriff

eigentlichen Buckets zugegriffen. Das Prinzip des Grid Directory ist damit sehr ähnlich der Arbeitsweise des dynamischen Hashing. Auch die Speichernutzung durch Mehrfachverweise auf dasselbe Bucket wird in Grid Files berücksichtigt. Dementsprechend ergeben sich ähnliche Vorgehensweisen bei der dynamischen Anpassung aufgrund dem Einfügen neuer Sätze. Läuft ein Bucket über, so muss er gesplittet werden. Verweisen zwei Verweise aus dem Grid-Directory auf den Bucket, kann eine wertabhängige Aufteilung erfolgen, das Directory muss bis auf die Verweise nicht verändert werden.

Das dynamische Verhalten eines Grid-Files lässt sich am besten anhand eines Beispiels durch eine Folge von Einfügungen. Der zwei-dimensionale Fall lässt sich graphisch besonders gut veranschaulichen. Die maximale Bucketkapazität liege in diesem Beispiel bei 3.

Ausgehend von einem leeren Grid-File wird in Abbildung 8.18 das Grid-File schrittweise gefüllt, wobei die zwei Dimensionen Abteilungsnummer und ProjektID einer Projektverwaltungsrelation als Beispielsituation dienen. Nach anfangs drei Einfügungen wird das erste Bucket gefüllt. Die vierte Einfügung erzwingt ein Splitten des Satzraums und die Zuweisung eines neuen Buckets. Die Wahl der Partitionierungsgrenze wird durch Überprüfung der vorhandenen Werte derart vorgenommen, dass eine möglichst gute Gleichverteilung gegeben ist. Der zweite Split zeigt, dass bei der Verfeinerung der Grid-Partition Bucketbereiche nicht betroffener Buckets neu aufgeteilt werden. Das bedeutet aber nicht, dass das zugehörige Bucket geteilt werden muss. Im Beispiel weisen zwei Pointer auf einen Bucket.

Dieses Prinzip funktioniert natürlich nicht nur bei zwei Dimensionen, auch bei drei oder mehr kann es angewendet werden, wobei eine graphische Darstellung des Prinzips aus nachvollziehbaren Gründen auf drei Dimensionen beschränkt ist (vgl. Abbildung 8.19).

Auf konzeptioneller Ebene ist die Anfragebearbeitung mit einem Grid-File recht einfach. Für jede spezifizierte Dimension wird der entsprechende Bereich im Grid-Directory betrachtet. Dies ist bei exakten Anfragen genau ein Bucket und kann bei Bereichsanfragen ein kompletter Bereich im Directory sein (Abbildung 8.20).

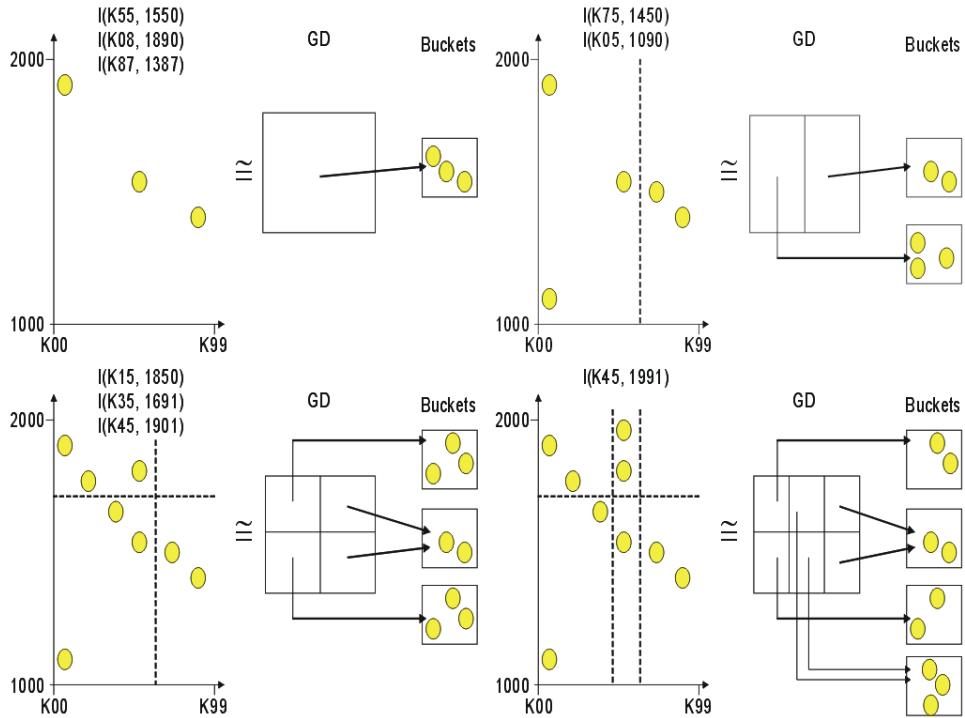


Abbildung 8.18: Dynamisches Verhalten eines Grid-Files

8.6 Weitere Verfahren

In der Literatur gibt es eine Vielzahl weiterer Verfahren, die zum großen Teil weder praktisch erprobt noch hinreichend allgemein sind, so dass die Beschreibung aller verschiedener Strukturen im Hinblick auf den praktischen Nutzen hier übertrieben wären. Deshalb sind nachfolgend einige der bedeutenderen Verfahren nur namentlich aufgeführt.

Quadrantenbaum: Unterstützung zweidimensionaler Zugriffe auf Punktobjekte aufgrund eines zusammengesetzten Schlüssels.

Mehrschlüssel-Hashing: Nutzung mehrerer Schlüssel beim Hashing.

k-d-Baum: Mehrdimensionaler binärer Baum, wobei die Baumebene die Nummer des verwendeten Schlüssels bestimmt.

Heterogener k-d-Baum: Zwischenknoten als reine Wegweiser.

k-d-B-Baum: Seitenorientierte Variante

hB-Baum: Variante des k-d-B-Baum

R-Baum: Zugriffspfad für räumliche Objekte

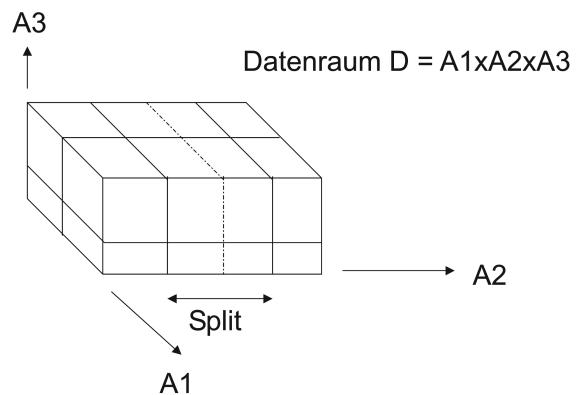


Abbildung 8.19: Dreidimensionaler Datenraum mit Zellpartition

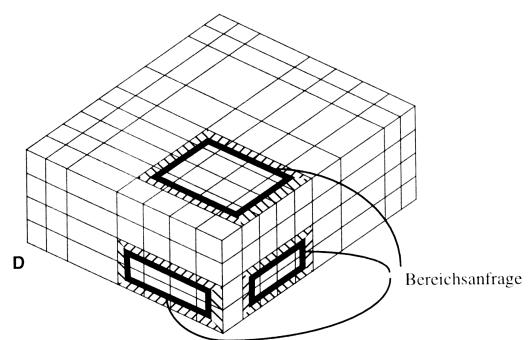


Abbildung 8.20: Grid-Directory-Erweiterung

Kapitel 9

Synchronisation

Eine Schlüsseleigenschaft von Datenbanksystemen ist, dass viele Benutzer gleichzeitig lesend und ändernd auf die gemeinsamen Datenbestände zugreifen können. Aufgabe der Synchronisation (Concurrency Control) ist es, die konkurrierenden Zugriffe voneinander zu isolieren, so dass die Konsistenz der Datenbank gewahrt und der Mehrbenutzerbetrieb gegenüber den Benutzern transparent bleibt (logischer Einbenutzerbetrieb).

Werden alle Transaktionen seriell ausgeführt, dann ist der geforderte logische Einbenutzerbetrieb ohne jegliche Synchronisation erreicht und die Datenbankkonsistenz wird durch den Mehrbenutzerbetrieb nicht gefährdet. Eine strikt serielle Ausführung der Transaktionen zur Umgehung der Synchronisationsprobleme verbietet sich jedoch vor allem aus Leistungsgründen, da hierbei selbst ein einziger Prozessor aufgrund langer Transaktionsunterbrechungen, beispielsweise wegen E/A-Vorgängen oder Denkzeiten bei Mehrschritttransaktionen, nicht effektiv genutzt werden kann.

Falls keine Synchronisation der Datenbankzugriffe durchgeführt wird, können verschiedene Konsistenzverletzungen (Anomalien) im Mehrbenutzerbetrieb eintreten. Um diese zu vermeiden und gleichzeitig eine hohe Leistungsfähigkeit zu erreichen, darf die Parallelität durch die Synchronisation möglichst wenig beeinträchtigt werden.

9.1 Anomalien im Mehrbenutzerbetrieb

Beim völlig unkontrollierten Zugriff auf Datenobjekte im Mehrbenutzerbetrieb können eine Reihe von unerwünschten Phänomenen auftreten, die bei einer seriellen Ausführung der Transaktion vermieden würden.

9.1.1 Verlorengegangene Änderungen (Lost Update)

Diese Anomalie tritt auf, wenn zwei Transaktionen gleichzeitig dasselbe Objekt ändern wollen und dabei eine der Änderungen verloren geht, indem sie durch die zweite überschrieben wird. Ein Beispiel für diese Anomalie zeigt Abbildung 9.1. Hier werden in zwei Transaktionen jeweils eine Gehaltsänderung für dieselbe Person durchgeführt. Dazu liest zunächst jede der Transaktionen den aktuellen Wert des Gehaltsattributs in eine lokale Programmvariable, mit der dann das erhöhte Gehalt bestimmt wird. Die von der ersten Transaktion zurückgeschriebene Änderung geht verloren, da die zweite Transaktion das Gehaltsattribut mit der von ihr berechneten Änderung überschreibt.

9.1.2 Lesen nicht freigegebener Änderungen (Dirty Read)

Geänderte Objekte, deren Änderung von Transaktionen stammen, die noch nicht beendet sind, werden auch als 'schmutzige Daten' bezeichnet. Da diese Transaktionen noch zurückgesetzt werden

Zeit	Gehaltsänderung 1	Gehaltsänderung 2
	SELECT GEHALT INTO :gehalt FROM PERS WHERE PNR = 656	
	gehalt := gehalt + 2000;	
	UPDATE PERS SET GEHALT = :gehalt WHERE PNR = 656	
		SELECT GEHALT INTO :gehalt FROM PERS WHERE PNR = 656
		gehalt := gehalt + 1000;
		UPDATE PERS SET GEHALT = :gehalt WHERE PNR = 656

Abbildung 9.1: Beispiel einer verlorengegangenen Änderung

können, ist die Dauerhaftigkeit der Änderung nicht gesichert. Eine Transaktion, die auf solche Änderungen zugreift, liest bzw. ändert somit ungültige Daten, wenn die Änderungstransaktion danach noch abgebrochen wird.

Zeit	Gehaltsänderung 1	Gehaltsänderung 2
	UPDATE PERS SET GEHALT = GEHALT + 1000 WHERE PNR = 546	
	...	SELECT GEHALT INTO :gehalt FROM PERS WHERE PNR = 546
	ROLLBACK	gehalt := gehalt * 1.1;
		UPDATE PERS SET GEHALT = :gehalt WHERE PNR = 546
		COMMIT

Abbildung 9.2: Beispiel eines Dirty Read

Ein Beispiel für einen solchen Lesezugriff zeigt Abbildung 9.2. Dabei wird im Rahmen einer Transaktion (Gehaltsänderung 2) das Gehalt eines angestellten in Abhängigkeit zu seinem früheren Gehalt berechnet. Die Transaktion 1 erhöht zuvor das Gehalt direkt. Da sich diese Transaktion jedoch aufgrund des ROLLBACK zurücksetzt, wird ihre Änderung rückgängig gemacht. Der Lesezugriff der zweiten Gehaltserhöhung war somit ein schmutziges Lesen, die Weiterverwendung des ungültigen Werts führt sogar zu inkorrekteten Folgeänderungen in der Datenbank.

Zu beachten ist, dass der Zugriff auf schmutzige Daten selbst dann zu Inkonsistenzen führen kann, wenn die Transaktion, deren Änderungen die Daten verschmutzt hat, nicht abgebrochen wird. Ein Beispiel hierfür ist der in Abbildung 9.3 gezeigte Fall einer sogenannten inkonsistenten Analyse, bei der eine statische Auswertung auf Basis unterschiedlicher Änderungszustände erfolgt. Dabei ändert Transaktion T1 die Gehälter von zwei Angestellten, während die Analysetransaktion T2 gleichzeitig die Gehaltssumme berechnet. Das ermittelte Ergebnis ist jedoch inkorrekt, da es die schmutzige Änderung des ersten Gehalts enthält, nicht jedoch die des zweiten Gehalts. Korrekt sind

Zeit	Gehaltsänderung T1	Gehaltssumme T2
	UPDATE PERS SET GEHALT = GEHALT + 1000 WHERE PNR = 546	
	...	
	UPDATE PERS SET GEHALT = GEHALT + 2000 WHERE PNR = 345	
	COMMIT	SELECT SUM(GEHALT) INTO :summe FROM PERS WHERE PNR IN (546, 345)

Abbildung 9.3: Inkonsistente Analyse aufgrund eines Dirty Read

dagegen nur Summenwerte, die auf einem von vollständig ausgeführten Transaktionen erzeugten Änderungszustand basieren, also vor oder nach Durchführung beider Gehaltserhöhungen.

9.1.3 Nicht-wiederholbares Lesen (Non-repeatable Read)

Diese Anomalie liegt vor, wenn eine Transaktion – bedingt durch Änderungen paralleler Transaktionen – während ihrer Ausführung unterschiedliche Werte eines Objekts sehen kann. Wenn etwa eine Transaktion das Gehalt eines Angestellten liest und danach eine parallel ausgeführte Transaktion dieses ändert, wird das erneute Lesen des Gehalts in der ersten Transaktion einen anderen Wert liefern. Die Transaktion sieht somit unzulässigerweise verschiedene Änderungsstände der Datenbank, was für Lesetransaktionen im Einbenutzerbetrieb nicht möglich ist.

Zeit	Lesetransaktion (Gehaltsdaten berechnen)	Änderungstransaktion
	SELECT AVG(GEHALT) INTO :gehalt FROM PERS	UPDATE PERS SET GEHALT = GEHALT + 1000 WHERE PNR = 546
	...	UPDATE PERS SET GEHALT = GEHALT + 2000 WHERE PNR = 439
	SELECT AVG(GEHALT + BONUS) INTO :gesamtgehalt FROM PERS	COMMIT

Abbildung 9.4: Inkonsistente Analyse ohne Dirty Read

Auch wenn man nicht explizit den selben Datensatz mehrfach liest, können verschiedene Datenbankabfragen dazu führen, dass ein Datensatz in der Ergebnisberechnung dieser unterschiedlicher Abfragen benötigt werden (siehe Abbildung 9.4).

9.1.4 Phantom-Problem

Eine besondere Form des nicht-wiederholbaren Lesens ist das Phantom-Problem. Dieses Problem ist zum einen dadurch bestimmt, dass in einer Lesetransaktion ein mengenorientiertes Lesen über ein bestimmtes Suchprädikat P erfolgt. Zum anderen wird durch eine gleichzeitig stattfindende Änderungstransaktion die Menge der sich für das Prädikat P qualifizierenden Objekte geändert. Das bedeutet, es gibt Phantomobjekte, die beispielsweise durch parallele Einfügungen oder Löschvorgänge plötzlich in der Ergebnismenge auftauchen bzw. aus ihr verschwinden. Damit kommt es bei einer erneuten Auswertung des Suchprädikats zu abweichenden nicht wiederholbaren Ergebnissen gegenüber dem ersten Zugriff.

Zeit	Lesetransaktion (Gehaltssumme überprüfen)	Änderungstransaktion (Neuen Mitarbeiter einfügen)
	<pre> SELECT SUM(GEHALT) INTO :Summe FROM PERS WHERE ANR = 17 ... SELECT GEHALTSSUMME INTO :GSumme FROM ABT WHERE ANR = 17 IF GSumme <> Summe </pre>	<pre> INSERT INTO PERS (PNR, ANR, GERHALT) VALUES (431, 17, 55000) UPDATE ABT SET GEHALTSSUMME = GEHALTSSUMME + 55000 WHERE ANR = 17 </pre>

Abbildung 9.5: Einfügen eines Phantomsatzes

Im Beispiel in Abbildung 9.5 wird die Gehaltssumme einer Abteilung bestimmt und mit dem separat gespeicherten Wert der Gehaltssumme verglichen, um bei einer Abweichung eine Inkonsistenzmeldung zu erzeugen. Nun wird nach der Selektion ein neuer Mitarbeiter korrekt eingefügt, die Gehaltssumme wird angepasst. Dieser Phantomsatz führt hier jedoch zu einer Fehlermeldung.

Die Vermeidung solcher Anomalien und die Wahrung der Datenbankkonsistenz im Mehrbenutzerbetrieb sind Aufgaben der Synchronisation. Hierzu ist eine präzisere Festlegung des Korrektheitskriteriums erforderlich, mit dem entschieden werden kann, ob eine bestimmte Verarbeitungsreihenfolge korrekt ist bzw. ob ein Synchronisationsalgorithmus korrekt arbeitet.

9.2 Korrektheitskriterium der Serialisierbarkeit

Das allgemein akzeptierte Korrektheitskriterium für die Synchronisation, welches die vorgestellten Anomalien ausschließt, ist die Serialisierbarkeit. Dieses Kriterium verlangt, dass die parallele Transaktionsausführung äquivalent ist zu einer seriellen Ausführungsfolge der beteiligten Transaktionen. Eine serielle Ausführung liegt vor, wenn keine Überlappung zwischen den Transaktionen vorliegt, sondern die Transaktionen vollständig nacheinander ausgeführt werden.

Eine parallele Ausführung von Transaktionen ist äquivalent zu einer seriellen, wenn für jeder der Transaktionen dieselbe Ausgabe wie in der seriellen Abarbeitungsreihenfolge abgeleitet wird und der gleiche Datenbankendzustand erreicht wird. Die Festlegung der Serialisierbarkeit als Korrektheitskriterium beruht auf der Tatsache, dass serielle und damit auch serialisierbare Transaktionsausführungen konsistenzhaltend sind und die angesprochenen Anomalien vermeiden. Demnach gilt es Synchronisationsverfahren zu entwickeln, die nur serialisierbare Transaktionsabläufe zulassen.

Als Operationen im Rahmen einer Transaktion werden im Allgemeinen nur Lese- und Schreiboperationen unterschieden. Unter Schreiben wird dabei das Ändern eines vorhandenen Objekts verstanden, nicht jedoch das Einfügen oder Löschen von Elementen. Unter einem Schedule versteht man eine Ablauffolge von Transaktionen mit ihren zugehörigen Operationen.

Eine zeitlich überlappende Ausführung von Transaktionen ist gemäß der gegebenen Definition serialisierbar und damit korrekt, wenn zu ihr ein äquivalenter serieller Schedule existiert. Die zur parallelen Transaktionsverarbeitung äquivalente serielle Ausführungsfolge wird auch als Serialisierungsreihenfolge bezeichnet. Diese Reihenfolge impliziert, dass eine Transaktion alle Änderungen der Transaktionen sieht, die vor ihr in der Serialisierungsreihenfolge stehen, jedoch keine der in dieser Reihenfolge nach ihr kommenden Transaktionen.

Um die Existenz eines solchen äquivalenten seriellen Schedule nachweisen zu können, müssen die zeitlichen Abhängigkeiten zwischen den Transaktionen berücksichtigt werden. Eine zeitliche Abhängigkeit entsteht dabei durch zwei Operationen verschiedener Transaktionen, die in Konflikt zueinander stehen, deren Ausführung also nicht reihenfolgenunabhängig ist. Im Falle von Lese- und Schreiboperationen liegt ein Konflikt zwischen zwei Operationen vor, wenn dasselbe Objekt angesprochen wird und mindestens eine der Operationen eine Schreiboperation darstellt. Man kann dabei Lese-Schreib-, Schreib-Schreib- und Schreib-Lese-Abhängigkeiten unterscheiden. Diese Abhängigkeiten können in einem Graphen dargestellt werden, in dem als Knoten die beteiligten Transaktionen stehen und die durch gerichtete Kanten die Abhängigkeiten zwischen Transaktionen repräsentieren. Es kann gezeigt werden, dass ein Schedule genau dann serialisierbar ist, wenn der zugehörige Abhängigkeitsgraph azyklisch ist. Denn nur in diesem Fall reflektiert der Graph eine partielle Ordnung zwischen den Transaktionen.

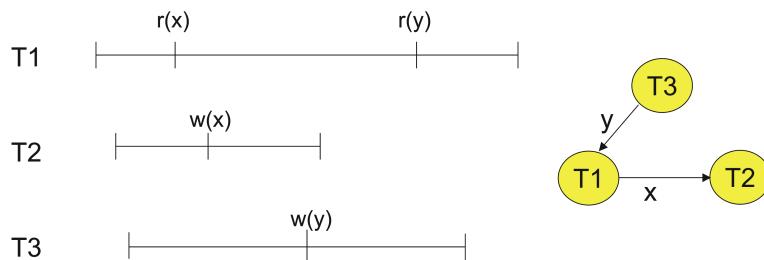


Abbildung 9.6: Schedule mit zugehörigem Abhängigkeitsgraphen

Abbildung 9.6 zeigt einen Schedule mit drei Transaktionen, wobei $r(x)$ bzw. $w(x)$ den Lese- bzw. Schreibzugriff der jeweiligen Transaktion auf Objekt x kennzeichnen. Der zugehörige Abhängigkeitsgraph keinen Zyklus enthält, ist der gezeigte Schedule serialisierbar, die Serialisierungsreihenfolge ist $T_3 < T_1 < T_2$.

Das Führen von Abhängigkeitsgraphen bietet keinen praktikablen Ansatz zur Implementierung eines Synchronisationsverfahrens, da hiermit meist erst nachträglich die Serialisierbarkeit von Schedules geprüft werden kann. Weiterhin wäre der Verwaltungsaufwand zu hoch. Zur Synchronisation greift man daher auf andere Verfahren zurück, für welche nachgewiesen werden konnte, dass sie Serialisierbarkeit gewährleisten.

9.3 Überblick zu Synchronisationsverfahren

Die historische Entwicklung von Synchronisationsverfahren in Datenbanksystemen erlaubt einen ersten Einblick auf die Vielfalt der vorgeschlagenen Synchronisationsalgorithmen. Das älteste und zugleich einfachste Verfahren, das aus Synchronisationstechniken in Betriebssystemen, wie Semaphore, hervorgegangen ist, kennt nur exklusive Objektsperren. Vor der Referenzierung eines

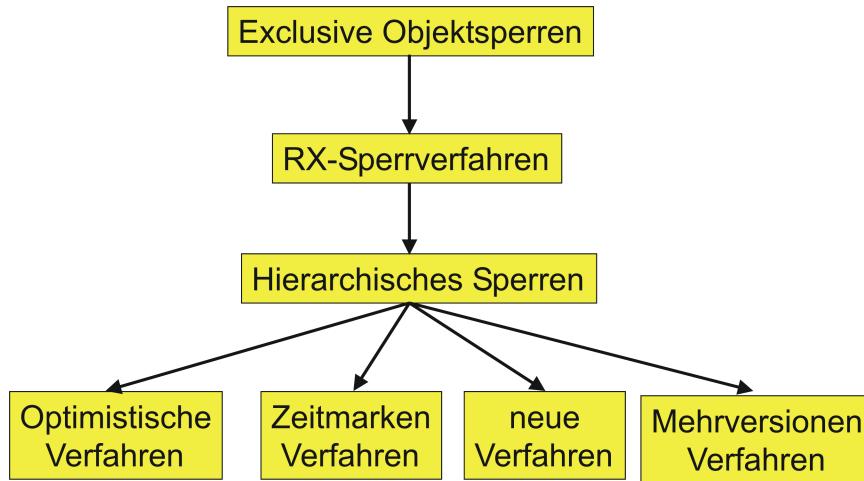


Abbildung 9.7: Historische Entwicklung von Synchronisationsverfahren

Objekts ist dabei eine exklusive Sperre zu erwerben, die alle anderen Transaktionen vom Zugriff auf das gesperrte Objekt ausschließt.

Da mit diesem primitiven Verfahren die angestrebte hohe Parallelität nicht erreicht werden kann, wurde in der nächsten Stufe der Entwicklung, dem RX-Verfahren, zwischen lesendem und schreibendem Objektzugriff unterschieden. Mit diesem, in heutigen Datenbanksystemen verbreiteten Sperrverfahren können mehrere Transaktionen parallel auf dasselbe Objekt lesend zugreifen. Eine Erweiterung des RX-Sperrprotokolls stellt die Verwendung von hierarchischen Objektsperren dar, bei der zwischen mehreren Sperrgranulaten und zusätzlichen Sperrmodi unterschieden wird. Solche hierarchischen Sperrverfahren werden in den meisten kommerziell verfügbaren Datenbanksystemen zumindest in einfacher Form (Sperrgranulate Satz und Tabelle) eingesetzt, da sie eine Eingrenzung des Synchronisationsaufwands gestatten.

Ausgehend von diesen Verfahren wurde eine Flut neuer oder abgewandelter Synchronisationsalgorithmen vorgestellt, die sich zumeist einer der im unteren Teil von Abbildung 9.7 angegebenen Gruppen zuordnen lassen. Zeitmarkenverfahren und optimistische Algorithmen stellen dabei zwei weitere Klassen allgemeiner Synchronisationstechniken dar. Im Gegensatz zu den optimistischen Ansätzen werden Sperrverfahren als pessimistisch bezeichnet. Diese Bezeichnung geht darauf zurück, dass Objekte vor jedem Zugriff gesperrt werden, nur um potentiell auftretende Konflikte zwischen Transaktionen behandeln zu können. Beim optimistischen Ansatz dagegen geht man davon aus, dass Konflikte relativ selten auftreten. Daher erfolgt zunächst ein unsynchronisierter Zugriff auf die Objekte. Erst am Transaktionsende wird geprüft, ob es zu Konflikten zwischen Transaktionen gekommen ist.

Neben den allgemeinen Synchronisationsalgorithmen wurden für besondere Leistungsprobleme spezielle Verfahren entwickelt. Dies betrifft unter anderem die Synchronisation auf Indexstrukturen und sogenannten Hot-Spot-Objekten, welche besonders häufig geändert werden.

9.4 Grundlagen von Sperrverfahren

9.4.1 Zwei-Phasen-Sperrverfahren

Sperrverfahren sind dadurch gekennzeichnet, dass vor dem Zugriff auf ein Objekt für die Transaktion eine Sperre zu erwerben ist. Dabei ist gemäß dem sogenannten Fundamentalsatz des Sperrens die Serialisierbarkeit gesichert, wenn folgende fünf Bedingungen eingehalten werden.

- ▶ Jedes zu referenzierende Objekt muss vor dem Zugriff mit einer Sperre belegt werden.
- ▶ Die Sperren anderer Transaktionen sind zu beachten. Das bedeutet, eine mit gesetzten Sperren unverträgliche Sperranforderung muss auf die Freigabe unverträglicher Sperren warten.
- ▶ Keine Transaktion fordert eine Sperre an, die sie bereits besitzt.
- ▶ Sperren werden zweiphasig angefordert und freigegeben.
- ▶ Spätestens bei Transaktionsende gibt eine Transaktion alle Sperren zurück.

Die Zweiphasigkeit bedeutet, dass eine Transaktion zunächst in einer Wachstumsphase alle Sperren anfordern muss bevor in der Schrumpfungsphase die Freigabe der Sperren erfolgt. Für dieses sogenannte Zwei-Phasen-Sperrprotokoll ergibt sich somit dass in Abbildung 9.8 gezeigte Verhalten hinsichtlich der Anzahl der Sperren einer Transaktion.

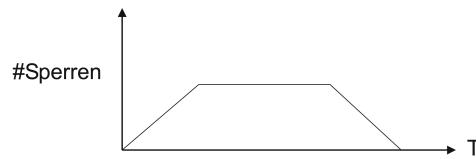


Abbildung 9.8: Zwei-Phasen-Sperren

Eine beim Fundamentalsatz zugrundeliegende Beschränkung ist jedoch die Annahme einer fehlerfreien Betriebsumgebung, in der Transaktionen nicht scheitern können und keine Systemausfälle vorkommen. Probleme aufgrund solcher Fehler ergeben sich dadurch, dass auch für Transaktionen in ihrer Schrumpfungsphase eine Rücksetzung notwendig werden kann. Zu diesem Zeitpunkt sind jedoch möglicherweise bereits Sperren freigegeben worden, so dass insbesondere Änderungen der Transaktion für andere Benutzer sichtbar wurden. Damit kommt es zu einem Dirty Read sowie daraus ableitbaren unzulässigen Folgeänderungen.

Eine prinzipielle Lösungsmöglichkeit besteht darin, das schmutzige Lesen von Änderungen einer Transaktion T zuzulassen, jedoch die Abhängigkeiten von Transaktionen, welche diese Änderungen lesen, zu vermerken. Dabei dürfen diese abhängigen Transaktionen nur dann erfolgreich beendet werden, wenn T nicht abgebrochen wird, sondern erfolgreich zu Ende kommt. Neben der aufwändigen Überwachung dieser Abhängigkeiten liegt ein Hauptproblem dieses Ansatzes darin, dass es kaskadierende Rücksetzungen impliziert. Denn das Rücksetzen von T führt nicht nur zum Abbruch aller direkt abhängigen Transaktionen T_i , die auf Änderungen von T zugegriffen haben, sondern auch von allen indirekt abhängigen Transaktionen, die auf Änderungen von T_i zugegriffen haben.

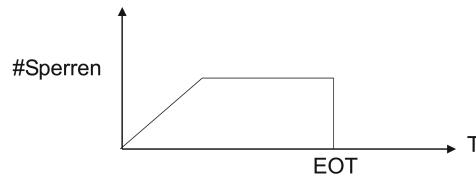


Abbildung 9.9: Striktes Zwei-Phasen-Sperren

Zur Vermeidung dieser Nachteile muss der Zugriff auf schmutzige Änderungen ausgeschlossen werden. Dies wird durch ein striktes Zwei-Phasen-Sperrprotokoll erreicht, bei dem die Schrumpfungsphase zur Sperrfreigabe vollständig zum Transaktionsende durchgeführt wird (Abbildung 9.9), wenn das Durchkommen der Transaktion sichergestellt ist. Bei erfolgreicher Beendigung einer Transaktion T erfolgt diese Sperrfreigabe im Rahmen des in Abbildung 9.10 dargestellten strikten

Zwei-Phasen-Commits. Hierzu werden die Sperren in der zweiten Commit-Phase freigegeben, nachdem in der ersten Commit-Phase das Überleben der Änderungen von T durch ein entsprechendes Logging gewährleistet wurde. Im Falle eines Scheiterns der Transaktion entfällt die Bestimmung und Rücksetzung von abhängigen Transaktionen, es kann ein isoliertes Rücksetzen von T erfolgen. Die Sperren werden freigegeben, nachdem T zurückgesetzt wurde.

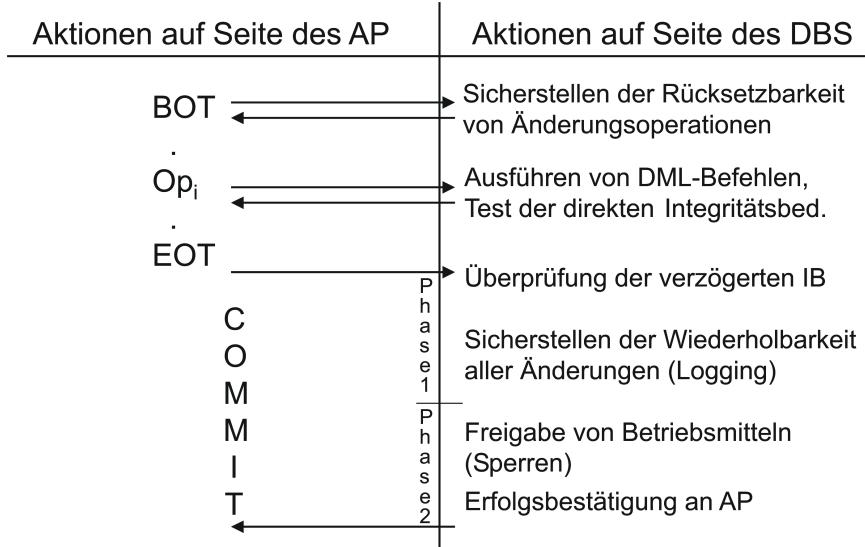


Abbildung 9.10: Zwei-Phasen-Commit

9.4.2 RX-Sperrverfahren

Das einfache RX-Sperrverfahren kennt nur zwei Sperrmodi, nämlich Lese- oder R-Sperren (read locks, shared locks) sowie Schreib- oder X-Sperren (exclusive locks). Die Verträglichkeit dieser Sperren zeigt folgende Tabelle:

		Aktueller Modus			
Anforderung		NL	R	X	
	R				
	X				

Die Verträglichkeitsangaben beziehen sich jeweils auf Zugriffe auf dasselbe Objekt. NL (No Lock) kennzeichnet die Situation, wenn ein Objekt von keiner laufenden Transaktion gesperrt ist, eine Sperranforderung wird in diesem Zustand stets bewilligt, wobei der aktuelle Modus des Objekts nach R bzw. X geändert wird. Die für Lesezugriffe erforderlichen R-Sperren sind miteinander verträglich, d.h. ein Objekt kann gleichzeitig von beliebig vielen Transaktionen gelesen werden. Die für Schreibzugriffe notwendigen X-Sperren sind dagegen weder mit sich selbst noch mit Lesesperrern kompatibel. Bei gesetzter X-Sperre sind somit alle weiteren Sperranforderungen abzulehnen. Ein Sperrkonflikt führt zu einer Blockierung der Transaktion, deren Sperranforderung den Konflikt verursacht hat. Diese Blockierungen können zu Deadlocks führen.

Die Aktivierung wartender Transaktionen erfolgt, sobald die unverträglichen Sperren freigegeben sind. Da das RX-Sperrverfahren ein paralleles Lesen und Ändern eines Objekts verbietet, kann einer Transaktion stets die aktuellste Version eines Objekts zur Verfügung gestellt werden und die Serialisierbarkeit ist gesichert.

Für den Schedule in Abbildung 9.6 tritt mit einem RX-Protokoll ein Sperrkonflikt auf. Für die Schreibsperrre von T2 auf Objekt x ergibt sich ein Konflikt aufgrund der zuvor gewährten R-Sperre für T1. T2 wird nach Beendigung und Freigabe der Sperren von T1 fortgesetzt. Analoges gilt für T1 und T3 für die Sperre auf y.

9.4.3 Behandlung von Sperrkonversionen

Hat beim RX-Verfahren eine Transaktion zunächst eine Lesesperrre für ein Objekt erworben und soll dieses Objekt geändert werden, ist eine Konversion der R- in eine X-Sperre erforderlich. Dies ist natürlich nur möglich, wenn keine anderen Transaktionen eine R-Sperre auf dem Objekt halten. Diese Konversionen führen jedoch häufig zu sogenannten Konversions-Deadlocks. Dabei handelt es sich um Situationen wie in Abbildung 9.11 gezeigt.

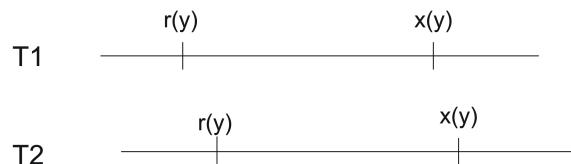


Abbildung 9.11: Konversionsdeadlock beim RX-Verfahren

Bei diesem Konversions-Deadlock wird ein Objekt y zunächst von zwei Transaktionen T1 und T2 mit einer R-Sperre belegt. Beide Transaktionen wollen danach das Objekt ändern, so dass sie eine Konversion in eine X-Sperre benötigen. Dabei ergibt sich jedoch eine zyklische Wartebedingung, da die X-Anforderung von T1 auf die Freigabe der R-Sperre von T2 warten muss und die X-Anforderung von T2 wegen der R-Sperre von T1 blockiert wird. Eine Lösung wäre, die R-Sperre freizugeben und statt einer Sperrkonversion eine X-Sperre anzufordern. Dieser Ansatz steht jedoch im Widerspruch zur Zweiphasigkeit und verletzt somit die Serialisierbarkeit.

Zur Umgehung dieses Problems implementieren einige kommerziell verfügbare Datenbanksysteme, wie beispielsweise IBM DB2, ein erweitertes Sperrverfahren mit drei Sperrmodi, dessen Kompatibilitätsmatrix wie folgt aussieht:

	NL	R	U	X
R				
U				
X				

Die neue Update-Sperre (U) wird dabei für Lesezugriffe mit Änderungsabsicht verwendet.¹ Eine U-Sperre kann zu einem Zeitpunkt höchstens einer Transaktion gewährt werden, sie ist jedoch mit bereits gewährten R-Sperren verträglich. Nachdem eine U-Sperre gesetzt wurde, werden gemäß Kompatibilitätsmatrix keine weiteren Leser mehr zugelassen, da sich ansonsten die X-Konversion unbestimmt lange Verzögerungen ergeben können. Soll eine Änderung erfolgen, wird eine Konversion der U- in eine X-Sperre vorgenommen. Dabei ist höchstens noch ein Warten auf die Freigaben solcher R-Sperren erforderlich, die bereits vor der U-Sperre gewährt wurden, so dass sich kein Deadlock bilden kann. Kommt es zu keiner Änderung, kann die U-Sperre in eine R-Sperre konvertiert werden (Downgrade), um andere Transaktionen das Lesen des Objekts bzw. das Anmelden einer Änderung zu gestatten. Insgesamt wird mit der U-Sperre wesentlich mehr Parallelität zugelassen, als wenn für Lesezugriffe mit Änderungsabsicht direkt eine X-Sperre gesetzt würde.

¹Um diese Sperren nutzen zu können, ist eine Erweiterung der SELECT-Anweisung vorzusehen, bei der die Selektion mit dem Zusatz FOR UPDATE versehen wird.

In dem Beispiel in Abbildung 9.11 fordern mit dem neuen Verfahren beide Transaktionen für den Lesezugriff eine U-Sperre an, die zunächst nur T1 gewährt wird, welche daraufhin die X-Konversion ohne Blockierung durchführen kann. Nach Ende von T1 wird die U-Anforderung von T2 bewilligt, so dass der Konversionsdeadlock umgangen wird.

9.4.4 Arbeiten mit Objektkopien

Um die Parallelität weiter zu erhöhen, legen Datenbanksysteme wie beispielsweise ORACLE eine Kopie des zu ändernden Objekts an. Hierdurch kann immer ein Lesevorgang durchgeführt werden, die mögliche Parallelität der Anwendungen steigt.

Für das Verfahren werden neben einer Lese-Sperre zwei weitere Sperrmodi benötigt. Eine A-Sperre wird gesetzt, wenn eine Objektkopie für die Änderungen angelegt wird. Dabei darf bei diesem Grundverfahren immer nur eine Kopie existieren, die A-Sperre ist mit sich selbst dadurch nicht verträglich. Wird eine Transaktion abgeschlossen, die eine A-Sperre auf ein Objekt besitzt, wird damit auch die Änderung in der Kopie gültig, die beiden Objektversionen müssen zusammengeführt werden. Dies wird mit der C-Sperre dargestellt, bei gesetzter C-Sperre existieren zwei gültige Objektversionen, eine zum Zeitpunkt vor der Änderung, eine zum Zeitpunkt nach der Änderung. Lesevorgänge können damit auch bei gesetzter C-Sperre durchgeführt werden, wobei je nach Start der Lesetransaktion die alte oder neue Version ausgelesen wird. Dies führt insbesondere dazu, dass das RAC-Verfahren nicht chronologieerhaltend ist, später gestartete Transaktionen bekommen alte Werte zu lesen.

Nach der Zusammenführung der beiden Objektversionen wird die C-Sperre vom Objekt entfernt. Dies führt zu folgender Kompatibilitätsmatrix:

	NL	R	A	C
R				
A				
C				

9.4.5 Hierarchische Sperrverfahren

Die Leistungsfähigkeit eines Synchronisationsverfahrens ist in hohem Maße von dem Synchronisationsgranulat abhängig, da es sowohl das Ausmaß an Konflikten als auch den Verwaltungsaufwand bestimmt. Werden Sperren auf groben Granulaten gesetzt, etwa einzelnen Tabellen oder gar der gesamten Datenbank, sind nur wenige Sperren zu setzen, der Verwaltungsaufwand ist somit gering. Allerdings wird es zu vielen Konflikten kommen, da große Teile der Datenbank blockiert sind. Dies ist besonders von Nachteil, wenn nur ein kleiner Teil der gesperrten Daten tatsächlich benötigt wird. Umgekehrt erlauben feine Sperrgranulate (einzelne Sätze) eine hohe Parallelität, verursachen jedoch einen hohen Overhead zur Wartung der Sperrtabelle. Wenn eine Transaktion etwa alle Sätze einer Relation auszuwerten hat, können bei Satzsperrn viele Tausende von Sperranforderungen notwendig werden, die einen erheblichen Bearbeitungsaufwand erfordern sowie die Sperrtabelle stark aufzublähen. In diesen Fällen wäre das Setzen einer einzigen Relationensperre vorzuziehen.

Es ist somit notwendig, mehrere Sperrgranulate zu unterstützen, um flexible Kompromisse hinsichtlich Parallelität und Verwaltungsaufwand zu ermöglichen. Insbesondere können dann kurze Transaktionen mit feinem Granulat synchronisiert werden (hohe Parallelität), während für komplexe Anfragen grobe Granularität (geringer Aufwand) wählbar sind. Die Realisierung eines derartigen Sperrverfahrens wird als hierarchisches Sperrverfahren bezeichnet, da die Sperrgranulate häufig eine Hierarchie der Art <Satz → Seite → (Relation, Index) → Datei → Datenbank> bilden. Die Hierarchiebildung kann zur Einsparung vieler Sperren und damit zur Reduzierung des Verwaltungsaufwands genutzt werden, da eine Lese- oder Schreibsperre auf einem Objekt

(beispielsweise einer Tabelle) zur Folge hat, dass alle Nachfolgeknoten in der Objekthierarchie (Sätze) implizit mitgesperrt werden und somit für diese keine eigenen Sperranforderungen zu stellen sind.

Kommerzielle Datenbanksysteme unterstützen zumeist eine mindestens zweistufige Objekthierarchie <Satz → Relation>. Bei der Ausführung von Transaktionen werden Sperren zunächst auf dem feinsten Granulat angefordert. Wird ein bestimmter Schwellwert an Sperren für eine Transaktion überschritten, erfolgt ein Umschalten auf das nächst gröbere Granulat. Dieser Vorgang wird als Sperreskalation bezeichnet. Für komplexe Anfragen werden direkt grobe Granulate gewählt.

Ein Problem bei der Verwendung mehrerer Sperrgranulate besteht darin, dass unverträgliche Zugriffe, die für Sperren auf unterschiedlichen Ebenen in der Objekthierarchie gesetzt werden, verhindert werden müssen. Wenn beispielsweise eine Transaktion T1 eine Schreibsperrre auf einer Relation R erwirbt, fordert sie auf den feineren Granulaten keine Sperren mehr an. Dennoch ist es zu verhindern, dass eine Transaktion T2 unbemerkt eine Satzsperrre für R bewilligt bekommt. Die Lösung dieses Problems besteht darin, dass neben expliziten Schreib- oder Lesesperrren auf einer bestimmten Granularitätsstufe alle Vorgängerknoten innerhalb der Sperrhierarchie durch sogenannte Anwartschaftssperrren (intention locks) zu sperren sind. Diese Anwartschaftssperrren zeigen die Absicht an, einen Zugriff auf einer tieferen Ebene vorzunehmen und verhindern unverträgliche Sperren auf den höheren Ebenen. Für ein RX-Sperrverfahren werden dabei zunächst zwei Arten von Anwartschaftssperrren, IR und IX unterschieden, deren Verträglichkeit Abbildung 9.12 zeigt.

	IR	IX	R	X
IR	+	+	+	-
IX	+	+	-	-
R	+	-	+	-
X	-	-	-	-

Abbildung 9.12: Einfaches Hierarchisches Sperrverfahren

Eine IR-Sperre (Intension Read) einer Transaktion signalisiert, dass auf einer feineren Granularitätsstufe eine R-Sperre folgt. Sie ist verträglich mit anderen Anwartschaftssperrren (IR oder IX) sowie expliziten R-Sperren.

Eine IX-Sperre (Intension eXclusive) signalisiert eine X-Sperre auf einer tieferen Stufe. Sie ist nur mit anderen Anwartschaftssperrren (IX oder IR) verträglich, nicht jedoch mit expliziten R- oder X-Sperren.

Das Setzen der Sperren muss von der Wurzel ausgehend nach unten in der Hierarchie erfolgen, die Freigabe der Sperren geschieht in umgekehrter Richtung. Bevor ein Knoten mit R oder IR gesperrt wird, sind alle Vorgänger in der Hierarchie für die betreffende Transaktion im IR-Modus zu sperren. Für eine X- oder IX-Sperre sind alle Vorgänger mit einer IX-Sperre zu belegen.

Das skizzierte Protokoll führt in einigen Fällen noch zu relativ vielen Behinderungen, insbesondere wenn in einem Satztyp zahlreiche Lesezugriffe, jedoch auch einige Änderungen vorzunehmen sind. Das Setzen einer X-Sperre in diesem Fall ist restriktiv und bedeutet eine erhebliche Beschränkung der Parallelität. Als Abhilfe für solche Situationen wurde die Verwendung eines zusätzlichen Sperrmodus, die RIX-Sperre vorgeschlagen. Eine RIX-Sperre entspricht dabei einer Kombination von R- und IX-Sperre. Das bedeutet, sie erlaubt ein Lesen des Objekts sowie aller Nachfolgeknoten, ebenso wie das Ändern von Nachfolgeknoten. Dabei sind nur für die Änderungen explizite X-Sperren erforderlich. Die Kompatibilität ist in Abbildung 9.13 dargestellt.

In der Kompatibilitätsmatrix ist zunächst die in Abschnitt 9.4.3 diskutierte U-Sperre enthalten, welche zur Vermeidung von Konversionsdeadlocks dient. Die U-Sperre, welche ein Lesen mit

	IR	IX	R	RIX	U	X
IR	+	+	+	+	-	-
IX	+	+	-	-	-	-
R	+	-	+	-	-	-
RIX	+	-	-	-	-	-
U	-	-	+	-	-	-
X	-	-	-	-	-	-

Abbildung 9.13: Verfeinertes Hierarchisches Sperrverfahren

Änderungsabsicht anzeigt, gestattet dem Sperrbesitzer ein Lesen des Objekts sowie der Nachfolgeobjekte. Zur Änderung erfolgt eine Konversion in eine X-Sperre. Das Setzen einer U-Sperre erfordert, ebenso wie eine X-, IX- oder RIX-Anforderung, dass alle Vorgängerknoten im RIX- oder IX-Modus gehalten werden.

Aufgrund der größeren Anzahl von Sperrmodi als beim RX-Verfahren ergeben sich vielfältige Möglichkeiten für Sperrkonversionen. Falls eine Transaktion zwei unterschiedliche Sperren für ein Objekt benötigt, ist jeweils das Maximum der Modi anzufordern bzw. eine bereits vorliegende Sperre in diesen Modus zu konvertieren. Hierbei wird die in Abbildung 9.14 gezeigte Ordnung zwischen den Sperrmodi berücksichtigt, wobei IR den schwächsten und X den restriktivsten Modus darstellt.

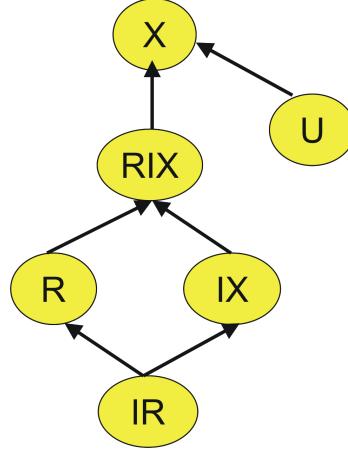


Abbildung 9.14: Sperrkonversionen

9.4.6 Logische Sperren

Üblicherweise werden Sperren auf physischen Datenbankobjekten wie Sätzen, Indexeinträge, Seiten oder Tabellen angefordert. Die Objekte verfügen über eindeutige Bezeichner, zu denen die Sperren innerhalb einer Sperrtabelle verwaltet werden. Physische Sperrverfahren haben jedoch Schwierigkeiten, das Phantom-Problem zu lösen, bei dem es für mengenorientierte Leseoperationen zu nicht-wiederholbaren Lesevorgängen aufgrund von gleichzeitigen Einfüge- oder Löschvorgängen kommen kann. Denn bei dem Lesezugriff können mit physischen Sperrverfahren im Allgemeinen nur die tatsächlich vorhandenen Objekte gesperrt werden, nicht jedoch diejenigen (Phantome), die nachträglich eingefügt werden. Somit kommt es nach dem Einfügen bei einer Neuauswertung der Leseoperation zu abweichenden Ergebnissen.

Eine elegante Lösungsmöglichkeit für das Phantom-Problem besteht in der Verwendung von so genannten logischen oder Prädikats-Sperren. Dabei verwenden Transaktionen zum Sperren die Operation Lock(R,P,a), wobei R die betroffene Relation bezeichnet, P ein logisches Prädikat sowie a den Zugriffswunsch (Read oder Write) angibt. Über das Prädikat kann dabei mit einer Sperre eine beliebige Menge von Objekten aus R angesprochen werden.

Zwei Sperranforderungen Lock(R,P,a) und Lock(R',P',a') stehen genau dann in Konflikt zueinander, wenn gilt:

- ▶ $R = R'$
- ▶ a oder a' ist keine Lesesperrre
- ▶ $P(t) \wedge P'(t) = \text{TRUE}$ für ein t aus R

Die Konflikterkennung verlangt somit, Prädikate auf Disjunkttheit ihrer Ergebnismengen zu überprüfen. Das Phantom-Problem wird umgangen, da nach einer Lesesperrre für das Prädikat P kein Einfügen von Objekten möglich ist, welche P erfüllen. Die Wiederholbarkeit des Lesevorgangs ist somit gewährleistet.

Das Problem ist jedoch, dass es im Allgemeinen unentscheidbar ist, ob zwei beliebige Prädikate disjunkte Mengen beschreiben. Eine Entscheidbarkeit lässt sich allerdings mit restriktiven Beschränkungen erreichen, was jedoch zu pessimistischen Entscheidungen und damit hoher Konfliktwahrscheinlichkeit führt. Der Extremfall P=TRUE entspricht der Verwendung von Relationensperren. Hierbei ist die Lösung des Phantom-Problems offensichtlich, da eine gesetzte Relationensperre das Einfügen oder Löschen von Sätzen für die Relation ausschließt. Jedoch werden durch das Sperren ganzer Relationen oft inakzeptabel viele Blockierungen verursacht.

Trotzdem existiert der Befehl **LOCK** (**Tabelle, Modus**) (also ohne Prädikat) bei vielen DBVS. Hierdurch kann ein Programmierer direkt eine Tabellensperre veranlassen, damit das DBS bei Transaktionen, die viele datensätze betreffen, nicht zunächst mit Satzsperrren beginnt, die dann im Verlauf auf eine Relationensperre umgesetzt werden müssen.

9.4.7 Deadlock-Behandlung

Eine mit Sperrverfahren eingehende Interferenz ist die Gefahr von Verklemmungen oder Deadlocks. Es handelt sich um ein allgemeines Problem bei der Betriebsmittelvergabe und ist durch das Zusammentreffen von fünf Voraussetzungen gekennzeichnet:

1. Paralleler Objektzugriff durch mehrere Transaktionen
2. Exklusive Zugriffsanforderungen
3. Die eine Sperre anfordernde Transaktion besitzt bereits Objekte/Sperren
4. Keine vorzeitige Freigabe von Objekten/Sperren
5. Zyklische Wartebeziehung zwischen zwei oder mehreren Transaktionen

Die charakterisierende Eigenschaft ist dabei die zyklische Wartebeziehung, die den weiteren Fortgang aller beteiligten Transaktionen verhindert. Die wechselseitigen Blockierungen führen oft dazu, dass weitere Transaktionen auf die blockierten Transaktionen warten müssen, so dass signifikante Leistungsprobleme eintreten können. Daher gilt es, Deadlocks möglichst rasch aufzulösen bzw. zu verhindern. Hierzu genügt es, dass eine der genannten Voraussetzungen wegfällt, wobei jedoch die ersten zwei Voraussetzungen im Datenbankbereich unumgänglich sind.

Prinzipiell gibt es vier generelle Ansätze zur Deadlock-Behandlung:

Deadlock-Verhütung: Die Entstehung von Deadlocks wird verhindert, ohne dass hierfür irgendwelche Maßnahmen während der Abarbeitung der Transaktion erforderlich sind. Hierfür wird meist verlangt, dass Transaktionen am Transaktionsbeginn alle benötigten Sperren anfordern. Da dies meist nicht möglich ist, sind dieser Ansatz für Datenbanksysteme nicht geeignet.

Deadlock-Vermeidung: Beim Eintreten eines Sperrkonflikts wird überprüft, ob die Blockierung der in Konflikt geratenen Transaktionen möglicherweise(!) einen Deadlock verursacht. Dies führt im positivem Fall direkt zum Zurücksetzen der Transaktion. Da diese Methode sehr ungenau ist, werden auch Transaktionen zurückgesetzt, die keinen Deadlock verursachen. Deshalb finden sich auch diese Verfahren in Datenbanksystemen nur selten.

Timeout-Verfahren: Bei diesem sehr einfachen und billigen Verfahren wird eine Transaktion zurückgesetzt, sobald ihre Wartezeit auf eine Sperre eine festgelegte Zeitschranke (beispielsweise 5 Sekunden) überschreitet. Die Anzahl der Rücksetzungen kann unnötigerweise sehr hoch werden. Dennoch wird es beispielsweise in Nonstop SQL eingesetzt.

Deadlock-Erkennung: Bei der Deadlock-Erkennung werden sämtliche Wartebeziehungen aktiver Transaktionen explizit in einem Wartegraphen protokolliert und Verklemmungen durch Zyklensuche in diesem Graphen erkannt. Die Auflösung eines Deadlocks geschieht durch Rücksetzung einer oder mehrerer am Zyklus beteiligter Transaktionen. Die Deadlock-Erkennung hat den großen Vorteil, dass nur bei tatsächlich vorliegenden Deadlocks Rücksetzungen stattfinden. Die Zyklensuche kann bei jedem Sperrkonflikt vorgenommen werden oder in periodischen Zeitabständen.

9.5 Konsistenzstufen

Die Unterstützung der Serialisierbarkeit ist zwar aus Korrektheitsgründen wünschenswert, kann jedoch aufgrund einer großen Anzahl von Sperrkonflikten erhebliche Leistungseinbußen verursachen. Denn bei einem strikten Zwei-Phasen-Sperrprotokoll sind sowohl Lese- als auch Schreibsperren bis zum Transaktionsende zu halten. Eine lange Sperrdauer erhöht zum einen die Wahrscheinlichkeit von Sperrkonflikten, zum anderen implizieren sie entsprechend lange Wartezeiten bis zur Aufhebung der Blockierung. Insbesondere können durch mengenorientierte Lesezugriffe erhebliche Probleme eingeführt werden, da sie häufig den Zugriff auf große Datenmengen sowie lange Ausführungszeiten erfordern (beispielsweise das Durchführen eines Relationen-Scans). Die zu setzenden Lesesperren können somit Änderungen für lange Zeit blockieren. Kommerzielle Datenbanksysteme unterstützen aus diesen Gründen häufig schwächere Korrektheitskriterien bzw. Konsistenzstufen als die Serialisierung unter Inkaufnahme bestimmter Mehrbenutzeranomalien.

9.5.1 Konsistenzstufen nach Gray

Wesentlich in diesem Zusammenhang ist die Unterscheidung von vier Konsistenzstufen, die zwischen 'langen' und 'kurzen' Sperren unterscheiden, wobei kurze Sperren nicht bis zum Transaktionsende, sondern nur für die Dauer der jeweiligen Datenbankoperation gehalten werden:

Konsistenzstufe 0: Die Transaktionen halten kurze Schreibsperren auf den Objekten, die sie ändern.

Konsistenzstufe 1: Transaktionen halten lange Schreibsperren auf den Objekten, die sie ändern.

Konsistenzstufe 2: Transaktionen halten lange Schreibsperren auf den Objekten, die sie ändern sowie kurze Lesesperren auf Objekten, die sie lesen.

Konsistenzstufe 3: Transaktionen halten lange Schreibsperrungen auf den Objekten, die sie ändern sowie lange Leseperren auf Objekten, die sie lesen.

Der Fall Konsistenzstufe 0 scheidet aus, da hier Lost Updates erfolgen können und dies eine Anomalie ist, die für einen praktikablen Einsatz auf jeden Fall verhindert werden soll. Ab Konsistenzstufe 1 werden Änderungen nicht mehr vor Transaktionende freigegeben, so dass das Überschreiben einer Änderung und damit typische Lost Updates vermieden werden. Da Leser in Stufe 1 jedoch keinerlei Sperrungen anfordern, können sie auf schmutzige Daten zugreifen (Dirty Read). Dies kann für Änderungstransaktionen nicht akzeptiert werden, da ihre Änderungen sonst auf inkonsistenten Daten aufbauen. Allerdings unterstützen kommerzielle Datenbanksysteme diese Konsistenzstufe – auch Browse-Modus genannt, für Lesezugriffe. Offenbar ist es zum schnellen Sichten von Daten vielfach akzeptabel, möglicherweise schmutzige Daten zu sehen. Dafür verursachen diese Lesezugriffe keinerlei Sperrkonflikte.

Konsistenzstufe 2 ist in der Praxis häufig vorzufinden. Durch das Setzen von Leseperren werden Dirty Reads ausgeschlossen. Da die Sperrungen kurz gehalten werden, kommt es zu weniger Sperrkonflikten und kürzeren Wartezeiten für Schreibanforderungen als mit langen Leseperren. Dafür sind jedoch nicht wiederholbare Lesevorgänge (Non-repeatable-Reads) in Kauf zu nehmen, da zwischen zwei Lesevorgängen eine andere Transaktion das Objekt ändern kann, wenn die Leseperren am Ende jeder Leseoperation freigegeben werden.

Konsistenzstufe 3 entspricht einem strikten Zwei-Phasen-Sperren und garantiert Serialisierbarkeit, wobei jedoch die Lösung des Phantom-Problems von der Realisierung des jeweiligen Sperrprotokolls abhängt.

9.5.2 Konsistenzstufen in SQL92

Der SQL92-Standard unterscheidet ebenfalls vier Konsistenzstufen (Isolation Level), jedoch mit anderer Definition als Gray. Um keine Vorgabe zur Implementierung zu treffen, erfolgte die Definition der Konsistenzstufen hinsichtlich der Anomalien, welche jeweils in Kauf genommen werden. Dabei wird sinnvollerweise verlangt, dass Lost Updates generell vermieden werden.

Konsistenzstufe	Dirty Read	Non-Rep-Read	Phantome
Read Uncommitted	+	+	+
Read Committed	-	+	+
Repeatable Read	-	-	+
Serializable	-	-	-

Die schwächste Isolationsstufe ist READ UNCOMMITTED, welche Konsistenzstufe 1 entspricht (Dirty Reads sind möglich, ebenso Non-Repeatable-Reads und Phantome). READ UNCOMMITTED ist in SQL92 nur für Lesetransaktionen zulässig. Die Stufe READ COMMITTED entspricht in etwa Konsistenzstufe 2, Lost Updates dürfen keine auftreten. Anstelle von Konsistenzstufe 3 werden jetzt zwei Stufen, nämlich REPEATABLE READ sowie SERIALIZABLE unterschieden. Dabei werden bei REPEATABLE READ Phantome nicht ausgeschlossen.

Die Wahl der Konsistenzstufe erfolgt transaktionsbezogen durch die Anweisung SET TRANSACTION, die folgendermaßen aufgebaut ist:

```
/-----\
SET TRANSACTION [ READ ONLY | READ WRITE] ISOLATION LEVEL ]
\-----/
```

Für den Transaktionsmodus bestehen zwei Alternativen: READ WRITE (Default), falls Änderungen möglich sein sollen, bzw. READ ONLY. Für den Level kann eine der vier Stufen aus der Tabelle angegeben werden, Default ist SERIALIZABLE.

9.6 Optimistische Synchronisation

Optimistische Synchronisationsverfahren gehen von der Annahme aus, dass Konflikte zwischen Transaktionen seltene Ereignisse darstellen und somit das präventive Sperren der Objekte unnötigen Aufwand verursacht. Daher greifen diese Verfahren zunächst nicht in den Ablauf einer Transaktion ein, sondern erlauben ein nahezu beliebig paralleles Arbeiten auf der Datenbank. Erst bei Transaktionsende wird überprüft, ob Konflikte mit anderen Transaktionen aufgetreten sind. Gemäß dieser Ausführung unterteilt man die Ausführung einer Transaktion in drei Phasen:

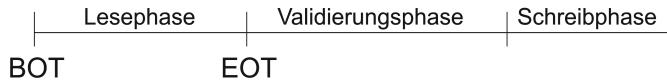


Abbildung 9.15: Ausführungsphasen bei optimistischer Synchronisation

1. In der Lesephase wird die eigentliche Transaktionsverarbeitung vorgenommen, Objekte der Datenbank werden gelesen und modifiziert. Jede Transaktion führt dabei ihre Änderungen auf privaten Kopien in einem ihr zugeordneten Transaktionspuffer durch, der für keine anderen Transaktionen zugänglich ist.
2. Beim Commit wird eine Validierungsphase gestartet, in der geprüft wird, ob die beendigungswillige Transaktion mit einer parallel zu ihr laufenden Transaktion in Konflikt geraten ist. Im Gegensatz zu Sperrverfahren, bei denen Blockierungen das primäre Mittel zur Behandlung von Synchronisationskonflikten sind, werden hier Konflikte stets durch Zurücksetzen einer oder mehrerer beteiligter Transaktionen aufgelöst. Es ist so mit mehr Rücksetzungen als bei Sperrverfahren zu rechnen, andererseits können bei optimistischen Verfahren keine Deadlocks entstehen.
3. Die Schreibphase wird nur von Änderungstransaktionen durchgeführt, welche die Validierungsphase erfolgreich beenden konnten. In dieser Phase wird zuerst die Wiederholbarkeit der Transaktion sichergestellt (Logging), bevor alle Änderungen durch Einbringen in die Datenbank für andere Transaktionen sichtbar gemacht werden.

Die Durchführung von Änderungen auf privaten Objektkopien bringt Vor- und Nachteile mit sich. Zum einen ist es ein allgemeiner Ansatz, andere Transaktionen vor schmutzigen Änderungen zu schützen. Es kann zeitgleich zur Änderung ein Lesezugriff auf die ungeänderte Version erfolgen, wodurch sich möglicherweise eine höhere Parallelität einstellt. Weiterhin sind Rücksetzungen von Transaktionen einfach zu realisieren, da hierzu lediglich die privaten Änderungen der Transaktion wegzwerfen sind, da diese noch nicht in der Datenbank sichtbar gemacht wurden. Andererseits verursachen die Kopien einen höheren Speicherbedarf sowie eine komplexere DB-Pufferverwaltung.

Um die Validierungen durchführen zu können, werden für jede Transaktion T_i während ihrer Lesephase die Namen von ihr gelesenen bzw. geänderter Objekte in einem Read-Set $RS(T_i)$ bzw. Write-Set $WS(T_i)$ geführt. Vor einer Objektänderung wird das Objekt gelesen, so dass der Write-Set einer Transaktion stets eine Teilmenge des Read-Sets bildet.

Optimistische Synchronisationsverfahren lassen sich gemäß ihrer Validierungsstrategie grob in zwei Klassen unterteilen. Bei den rückwärtsorientierten Verfahren (Backward Oriented Optimistic Concurrency Control, BOCC) erfolgt die Validierung ausschließlich gegenüber bereits beendeten Transaktionen. Bei den vorwärtsorientierten Verfahren (Forward Oriented Optimistic Concurrency Control, FOCC) dagegen wird gegen noch laufende Transaktionen validiert. In beiden Fällen wird durch die Validierung sichergestellt, dass die validierende Transaktion alle Änderungen von zuvor erfolgreich validierten Transaktionen gesehen hat. Damit ist die Serialisierungsreihenfolge durch die Validierungsreihenfolge gegeben.

Im ursprünglichen BOCC-Verfahren wird bei der Validierung überprüft, ob die validierende Transaktion ein Objekt gelesen hat, das während ihrer Lesephase geändert wurde. Dazu wird in der Validierungsphase der Read-Set der validierenden Transaktion T mit dem Write-Set aller Transaktionen T_i verglichen, die während der Lesephase von T validiert haben.

```
VALID = true;
for (alle während T-Ausführung beendeten  $T_i$ ) do
    if RS( $T$ )  $\cap$  WS( $T_i$ ) then VALID := FALSE;
endfor;
if VALID THEN Schreibphase für  $T$ 
else Rollback ( $T$ )
```

Ergibt sich eine Überschneidung mit einem dieser Write-Sets, wird die validierende Transaktion zurückgesetzt, da sie möglicherweise auf veraltete Daten zugegriffen hat. Die am Konflikt beteiligten Transaktionen können nicht mehr zurückgesetzt werden, da sie bereits beendet sind. Die Validierungen werden dabei in einem kritischen Abschnitt durchgeführt, der sicherstellt, dass zu einem Zeitpunkt höchstens eine Validierung vorgenommen wird.

Die skizzierte BOCC-Validierung hat den Nachteil, dass Transaktionen oft unnötigerweise (wegen eines unechten Konflikts) zurückgesetzt werden, obwohl die aktuellen Objektversionen gesehen wurden. Dies ist dann der Fall, wenn auf das von einer parallelen Transaktion geänderte Objekt erst nach dem Einbringen in die Datenbank zugegriffen wurde. Eine Abhilfe des Problems wird jedoch möglich, indem man Änderungszähler oder Versionsnummern an den Objekten führt und eine Rücksetzung nur vornimmt, wenn tatsächlich veraltete Daten gelesen wurden. Dieser Ansatz wird als BOCC+ bezeichnet.

Ein schwerwiegenderes Problem ist jedoch die Gefahr des Verhungerns, dass also Transaktionen bei der Validierung ständig scheitern. Dies ist vor allem für lange Transaktionen zu befürchten, da sie einen großen Read-Set aufweisen und sich gegenüber vielen Transaktionen validieren müssen. Weiterhin verursacht das späte Rücksetzen am Transaktionsende ein hohes Maß unnötig verrichteter Arbeit.

Diese Probleme werden in FOCC-Verfahren abgeschwächt. Bei ihnen erfolgt die Validierung nicht gegen bereits beendete Transaktionen, sondern gegenüber aktiven Transaktionen. In der Validierungsphase, die nur von Änderungstransaktionen durchzuführen ist, wird untersucht, ob eine der in der Lesephase befindlichen Transaktionen ein Objekt gelesen hat, das die validierende Transaktion zu ändern im Begriff ist.

```
VALID = true;
for (alle laufenden  $T_i$ ) do
    if RS( $T_i$ )  $\cap$  WS( $T$ ) then VALID := FALSE;
endfor;
if VALID THEN Schreibphase für  $T$ 
else Rollback ( $T_{Konfliktpartei}$ )
```

In Konfliktfall muss der Konflikt durch Zurücksetzen einer (oder mehrerer) der beteiligten Transaktionen aufgelöst werden. Anstatt der validierenden Transaktion können also auch die betroffenen laufenden Transaktionen zurückgesetzt werden, um beispielsweise den Arbeitsverlust zu verringern. Auch das Verhungern von Transaktionen kann verhindert werden, indem beispielsweise bei der Auswahl der Opfer die Anzahl der bereits erfolgten Rücksetzungen berücksichtigt wird. Im Gegensatz zum BOCC-Ansatz führen bei FOCC daneben nur echte Konflikte zu Rücksetzungen.

Optimistische Verfahren haben gegenüber Sperrverfahren den Vorteil der Deadlock-Freiheit sowie einer potentiell höheren Parallelität, da Transaktionen nicht aufgrund von Sperrkonflikten blockiert werden. Auf der anderen Seite kann es zu einer hohen Anzahl von Rücksetzungen kommen, da dies die einzige Methode der Konfliktbehebung darstellt. Diese Gefahr besteht vor allem für lange Transaktionen sowie für Zugriffe auf häufig geänderte Datenobjekten.

Jedoch ist es möglich optimistische und pessimistische Synchronisation zu kombinieren, wobei solche hybriden Ansätze schon Einzug in objektorientierte Datenbankprodukte und Borlands

Paradox gefunden hat. Bei den kombinierten Verfahren erfolgt die Wahl zwischen optimistischer und pessimistischer Synchronisation entweder auf der Ebene von Transaktionen oder auf der Ebene von Objekten. Erstere Methode ist beispielsweise sinnvoll um mit einer pessimistischen Synchronisation einer langen oder bereits gescheiterten Transaktion ein Durchkommen zu sichern, während der zweite Ansatz vor allem bei Hot-Spot-Objekten angebracht ist, die aufgrund häufiger Änderungen viele Konflikte verursachen. Ein inhärenter Nachteil von solch hybriden Ansätzen ist der hohe Realisierungsaufwand beide Verfahrensarten zu unterstützen ohne möglicherweise die Vorteile beider Ansätze tatsächlich zu erreichen.

9.7 Leistungsbewertung von Synchronisationsverfahren

Die Diskussion in den vorangegangenen Abschnitten verdeutlichte die große Anzahl möglicher Synchronisationsverfahren, wobei nur ein Bruchteil der in der Literatur behandelten Ansätze angesprochen werden konnte. Ebenso gibt es eine nahezu unüberschaubare Anzahl von Publikationen mit Leistungsanalysen einzelner Verfahren. Dabei werden vorwiegend analytische Modelle sowie Simulationsmethoden eingesetzt, jedoch mit sehr starken Abweichungen bei der Modellierung sowie im Detailierungsgrad. Nicht selten wurden daher auch sich widersprechende Leistungsaussagen gewonnen. Nachfolgend werden nur unstrittige Aussagen dargestellt.

Die mit Synchronisationsverfahren eingeführten Blockierungen und Rücksetzungen können die Leistungsfähigkeit eines Datenbanksystems, insbesondere Durchsatz und Antwortzeit, erheblich beeinflussen und zu einem gravierenden Engpass werden. Die Leistungsfähigkeit eines bestimmten Verfahrens ist dabei von zahlreichen Faktoren beeinflusst, insbesondere der jeweiligen Transaktionslast, der Datenbankstruktur und -größe, der Hardware- und Datenbankkonfiguration. Besondere Bedeutung kommt dabei dem Multiprogramming Level (MPL) zu, also der Anzahl gleichzeitig aktiver Transaktionen. Für Sperrverfahren, aber auch für andere Synchronisationsverfahren ergibt sich dabei tendenziell das in Abbildung 9.16 gezeigte Verhalten.

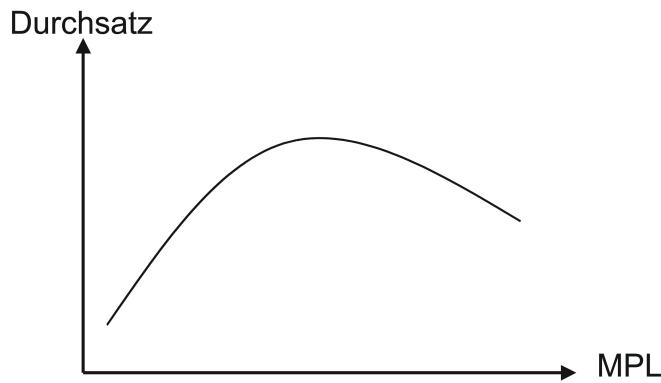


Abbildung 9.16: Einfluss des MPL auf den Durchsatz

Zunächst bewirkt ein Erhöhen des MPL den gewünschten Anstieg des Durchsatzes, da die verfügbaren Ressourcen, insbesondere die Prozessoren, besser ausgelastet werden können. Jedoch kommt es dabei auch zunehmend zu Behinderungen, so dass ab einem gewissen MPL keine weitere Durchsatzsteigerung mehr erzielt werden kann. Vor allem bei Sperrverfahren kann es sogar zu Trashing-Effekten kommen, so dass sich bei weiterer MPL-Steigerung der Durchsatz verringert, denn das Ansteigen des MPL bedeutet eine entsprechende Zunahme an gleichzeitig gesetzten Sperren und somit wachsende Konfliktgefahr. Zugleich erhöht sich aufgrund der Blockierungen die Bearbeitungszeit von Transaktionen, was zu einer weiteren Steigerung der Konfliktrate führt. Im Sättigungsbereich hinzukommende Transaktionen erwartet nicht nur ein hohes Konfliktrisiko,

sondern sie verursachen auch zunehmende Konflikte für die bereits laufenden Transaktionen, was zu einer Abnahme des Durchsatzes beiträgt.

Für optimistische Synchronisationsverfahren sowie anderen mehr auf Rücksetzungen basierende Verfahren kommt es mit wachsendem MPL ebenfalls zu einer entsprechenden Zunahme an Konflikten. Dabei ist der Arbeitsverlust durch das Abbrechen von Transaktionen in der Regel nachteiliger als das Warten auf eine Sperre. Hinzu kommt das inhärente Problem des wiederholten Zurücksetzens derselben Transaktion, so dass bestimmte Transaktionen im Extremfall verhungern können. Diese Effekte ergaben in mehreren detaillierten Leistungsstudien eine Unterlegenheit gegenüber Sperrverfahren. Nur in nahezu konfliktfreien Umgebungen wurden ähnlich gute Leistungsmerkmale erzielt.

In existierenden Datenbanksystemen trägt auch der Systemverwalter eine große Mitverantwortung für die Konflikthäufigkeit. Denn üblicherweise ist der maximale MPL ein von ihm manuell einzustellender Systemparameter, der wie diskutiert die Leistungsfähigkeit maßgeblich beeinflusst. Insbesondere muss eine Einstellung gefunden werden, die Trashing vermeidet und dennoch einen hohen Durchsatz ermöglicht. Diese Aufgabe ist aufgrund der üblicherweise starken Schwankungen unterworfenen Lastsituation nur schwer lösbar, so dass auch zur Erleichterung der Systemverwaltung ein dynamischer Ansatz zur MPL-Kontrolle von großer Wichtigkeit ist.

Kapitel 10

Datenbankrecovery

Eine sehr wichtige Aufgabe von Datenbanksystemen liegt in der Gewährleistung einer weitgehenden Datensicherung. Trotz Auftretens von Fehlern verschiedener Art ist die Konsistenz der Datenbank automatisch zu wahren und Datenverlust zu verhindern. Die Fehlerbehandlung ist Aufgabe der Recovery-Komponente des Datenbanksystems. Sie benötigt neben den Datenbankinhalten redundante Informationen, welche durch ein Logging im Normalbetrieb zu protokollieren sind. Die notwendigen Recovery-Aufgaben sind weitgehend durch das Transaktionskonzept (vor allem die Eigenschaft A und D des ACID-Prinzips) bestimmt. Insbesondere sind aufgrund der Dauerhaftigkeitszusicherung Änderungen erfolgreich beendeter Transaktionen gegenüber allen erwarteten Fehlerarten zu bewahren. Weiterhin verlangt die Alles-oder-Nichts-Eigenschaft das Zurücksetzen von Änderungen für Transaktionen, welche aufgrund eines Fehlers ihr Commit nicht abschließen konnten.

10.1 Fehler- und Recovery-Arten

Datenbanksysteme müssen üblicherweise drei Fehlerarten behandeln können: Transaktionsfehler, Systemfehler sowie Geräte- bzw. Externspeicherfehler. Jede dieser Fehlerklassen verlangt entsprechende Recovery-Maßnahmen. Hierfür werden die in Abbildung 10.1 gezeigten und an der Fehlerbehandlung beteiligten Systemkomponenten benötigt.

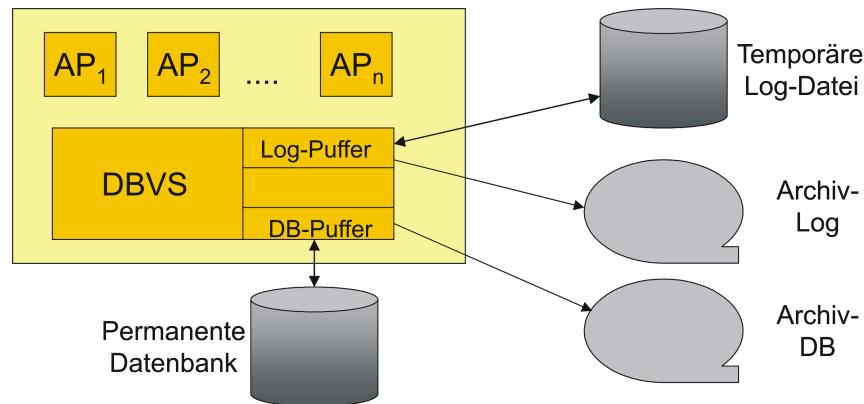


Abbildung 10.1: Systemkomponenten

Den auf Externspeicher vorliegenden Teil der Datenbank bezeichnet man als permanente oder materialisierte Datenbank. Teile der Datenbank befinden sich daneben im Datenbankpuffer im

Hauptspeicher, insbesondere Änderungen, die noch nicht in die permanente Datenbank zurückgeschrieben wurden. Zur Fehlerbehandlung wird eine Log-Datei auf dedizierten Externspeichern geführt, in der alle zuletzt ausgeführten Datenbankänderungen protokolliert werden. Die Log-Sätze werden üblicherweise im Hauptspeicher innerhalb eines Log-Puffers gesammelt, der zu bestimmten Zeitpunkten (beispielsweise beim Commit) ausgeschrieben wird. Die Log-Datei wird gelegentlich als temporäre Log-Datei bezeichnet, da in ihr schon aus Platzgründen die Einträge nur begrenzte Zeit aufbewahrt werden. Ältere Log-Sätze werden innerhalb von Archiv-Logs geführt. Diese enthalten alle Änderungen seit Erstellung einer bestimmten Archivkopie (Backup), die einen älteren Schnappschuss der permanenten Datenbank darstellt. Archivkopien und Archivlogs dienen zur Behandlung von Externspeicherfehlern und werden wegen ihres hohen Speicherbedarfs und ihrer relativ seltenen Nutzung oft auf preiswerten und langsamen Medien wie Bandspeicher vorgehalten.

10.1.1 Transaktionsfehler

Die mit Abstand häufigste Fehlerart sind Transaktionsfehler, bei denen lediglich eine Transaktion oder einige wenige Transaktionen betroffen sind. Beispiele für das Auftreten solcher Fehler sind:

- ▶ freiwilliger Transaktionsabbruch durch eine Rollback-Anweisung (beispielsweise aufgrund unzuverlässiger Dateneingabe oder nicht erfolgreicher Datenbankoperation)
- ▶ Fehler im Anwendungsprogramm
- ▶ systemseitiger Abbruch einer Transaktion beispielsweise aufgrund einer Verletzung von Integritätsbedingungen oder Zugriffsbeschränkungen
- ▶ systemseitiger Abbruch von einer oder mehreren Transaktionen zur Auflösung von Verklemmungen, Behandlung von Systemüberlast (beispielsweise bei Sperrengässen), aufgrund einer geplanten Systemschließung, usw.

Die Behandlung solcher Transaktionsfehler verlangt aufgrund der Alles-oder-Nichts-Eigenschaft das isolierte Zurücksetzen der betroffenen Transaktion(en) im laufenden Betrieb. Die schnelle Ausführung dieser UNDO-Recovery ist aufgrund der relativen Häufigkeit von Transaktionsfehlern besonders wichtig. Zudem sind die Behinderungen für andere Transaktionen durch das rasche Freigeben von Sperren und andere Ressourcen möglichst gering zu halten. Die Durchführung der Transaktionsfehler-Recovery kann mit der aktuellen Datenbank – insbesondere auch mit den im Hauptspeicher gepufferten Datenbankseiten – sowie den Log-Sätzen der Transaktionen auf der entsprechenden Log-Datei (bzw. im Log-Puffer) erfolgen.

Für lange Transaktionen ist das vollständige Zurücksetzen auf den Transaktionsbeginn oft mit einem hohen Arbeitsverlust verbunden. Eine Abhilfemöglichkeit besteht in der Verwendung transaktionsinterner Rücksetzpunkte (Savepoints), um erreichte Zwischenstände zu sichern. Damit kann im Fehlerfall ein partielles Zurücksetzen der Transaktion auf einen Rücksetzpunkt erfolgen. Dies wird in Abschnitt 5.6.3 näher behandelt.

10.1.2 Systemfehler

Ein Systemfehler liegt vor, wenn der weitere Betrieb des Datenbanksystems nicht mehr möglich ist. Dies kann durch Fehler der Hardware (beispielsweise Rechnerausfall) oder Software (Absturz) sowie durch Umgebungsfehler (beispielsweise Stromausfall) verursacht sein. Wesentlich dabei ist, dass die Hauptspeicherinhalte verloren gehen. Damit sind alle Änderungen, welche zum Fehlerzeitpunkt nur im Hauptspeicher vorlagen, verloren und müssen gegebenenfalls rekonstruiert werden.

Die Behandlung von Systemfehlern erfolgt durch die sogenannte Crash-Recovery, von der alle zum Fehlerzeitpunkt laufenden Transaktionen betroffen sind. Hierbei wird davon ausgegangen, dass

die auf Externspeicher vorliegende permanente Datenbank nicht zerstört ist, wenngleich sie im Allgemeinen in einem inkonsistenten Zustand vorliegt. Ziel der Crash-Recovery ist die Herstellung des jüngsten transaktionskonsistenten Datenbankzustands. Hierzu sind

- ▶ im Rahmen einer UNDO-Recovery Änderungen von nicht erfolgreich zu Ende gekommenen Transaktionen, welche vor dem Fehler in die permanente Datenbank gelangten, zurückzusetzen sowie
- ▶ durch eine REDO-Recovery für erfolgreich beendete Transaktionen deren Änderungen zu wiederholen, falls sie aufgrund des Systemfehlers noch nicht in die permanente Datenbank gelangten.

Die Durchführung dieser Recovery-Aktionen erfolgt mit der permanenten Datenbank sowie der temporären Log-Datei.

10.1.3 Geräte- bzw. Externspeicherfehler

Externspeicherfehler betreffen vor allem den Ausfall von Magnetplatten. Die Behandlung solcher Fehler ist offensichtlich sehr wichtig, um die permanente Datenbank weiter nutzen zu können und Datenverlust zu verhindern. Aufgabe der Geräte- bzw. Plattenrecovery ist somit vor allem eine REDO-Recovery zur Rekonstruktion der durch den Ausfall verloren gegangenen Änderungen. Hierzu sind regelmäßig Archivkopien der Datenbank anzulegen. Nach dem Ausfall einer Platte der permanenten Datenbank wird die letzte Archivkopie zunächst eingespielt. Daraufhin werden alle Änderungen erfolgreicher Transaktionen, die seit Erstellung der Archivkopie durchgeführt wurden, mit dem Archiv-Log ergänzt, um den aktuellsten, d.h. den jüngsten transaktionskonsistenten Datenbankzustand zu erreichen.

10.1.4 Katastrophen-Recovery

Weitgehende Probleme entstehen, wenn beispielsweise ein ganzes Rechenzentrum aufgrund einer Naturkatastrophe (Erdbeben, Überschwemmung) oder eines Terroranschlags zerstört wird. In diesem Fall sind sowohl die Verarbeitungsrechner als auch die Externspeicher betroffen, einschließlich der am gleichen Ort geführten Log-Dateien und Archivkopien. Im Falle solcher Katastrophen kann ein Datenverlust nur über einen verteilten Systemansatz verhindert werden, bei dem die Daten an zwei oder mehr geographisch weit entfernten Knoten repliziert gespeichert werden. Dabei sind durchgeführte Änderungen über das Kommunikationsnetz ständig zwischen den beteiligten Knoten auszutauschen um die Kopien auf dem neuesten Stand zu halten. Die sicherste Lösung sieht dabei vor, dass eine Änderungstransaktion erst dann erfolgreich abgeschlossen wird, wenn auch die geographisch entfernte Datenbankkopie aktualisiert wurde.

10.1.5 Grenzen der Recovery

Die genannten Fehlerklassen decken ein weites Spektrum ab, so dass mit den entsprechenden Recovery-Arten ein umfangreicher Schutz erreicht werden kann. Allerdings geht die Realisierung der Fehlerbehandlung von einigen grundlegenden Annahmen aus, welche in der Praxis nur bedingt zutreffen. Bei Verletzung der Annahmen kann somit die Korrektheit der Fehlerbehandlung und damit die Korrektheit der Datenbank nicht mehr gewährleistet werden.

Eine derartige Voraussetzung ist, dass das Datenbankmanagementsystem selbst als fehlerfrei angesehen wird. Dies ist natürlich schon aufgrund des Umfangs und der Komplexität solcher Systeme unrealistisch. Wenn nun aber beispielsweise bei der Implementierung der Logging- und Recovery-Funktionen Fehler vorliegen, sind unvorhersehbare und beliebig weitreichende Konsistenzverletzungen der Datenbank möglich.

Weiterhin geht das Datenbanksystem von der logischen Korrektheit von Transaktionsprogrammen aus. Wird nachträglich festgestellt, dass bestimmte Änderungen aus Sicht der Anwendungslogik unzutreffenderweise erfolgten, kann hierfür keine automatische Rücknahme erfolgen. Hier sind manuelle Korrekturen vorzunehmen.

Schließlich setzt die Korrektheit der Fehlerbehandlung auch eine ordnungsgemäße Systemverwaltung voraus, u.a. zur Erstellung von Archivkopien.

10.2 Logging-Techniken

Zur Durchführbarkeit der Recovery sind seitens des Datenbanksystems die Datenbankänderungen zu protokollieren. Wie in Abbildung 10.2 schematisch dargestellt, werden hierbei für jede Änderung im Normalbetrieb (DO-Operationen) entsprechende Log-Sätze gesichert, welche die durchgeführte Transformation des Datenbankzustands beschreiben. Im Falle einer REDO-Recovery wird ausgehend vom ungeänderten Datenbankzustand mit dem Log-Satz der neue Zustand rekonstruiert. Umgekehrt wird für die UNDO-Recovery ausgehend von dem geänderten Datenbankzustand mit der Log-Information der alte Zustand vor der Änderung wiederhergestellt. Einige Systeme schreiben spezielle Log-Sätze, sogenannte Compensation Log Records (CLR), um Fehler, die während des Recovery auftreten, korrekt zu behandeln.

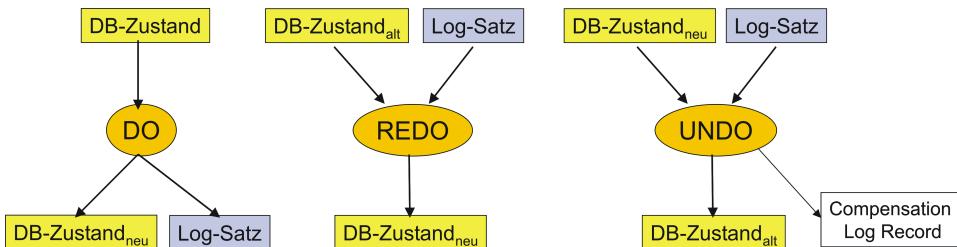


Abbildung 10.2: DO-REDO-UNDO-Prinzip

Für das Logging bestehen dabei mehrere generelle Alternativen, die in Abbildung 10.3 eingeordnet sind.

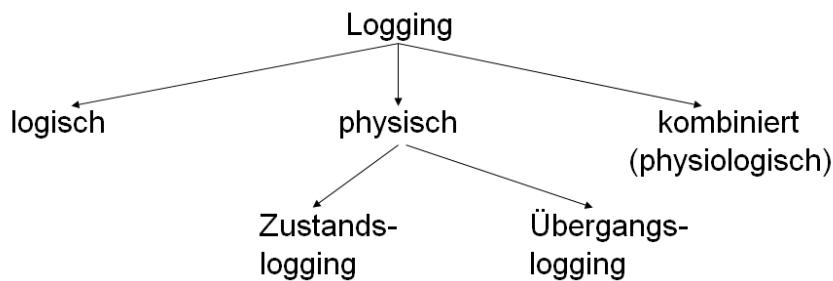


Abbildung 10.3: Klassifikation von Logging-Verfahren

10.2.1 Logisches Logging

Beim Logischen Logging werden die durchgeführten Änderungsoperationen mit ihren Parametern protokolliert. Zum Beispiel würde für das Einfügen eines neuen Angestellten lediglich die zugehörige Insert-Operation mit den Attributwerten in den Log geschrieben werden. Die Protokollierung

von Änderungen in Seiten und Indexstrukturen ist nicht erforderlich, so dass das logische Logging eine sehr elegante Lösung mit minimalem Log-Umfang verspricht.

Allerdings zeigen sich bei näherer Betrachtung große Schwierigkeiten bei der Realisierung logischer Log-Strategien. Ein Hauptproblem liegt darin, dass im Fehlerfall (insbesondere nach einem Rechnerausfall) vorliegende materialisierte Datenbank die Ausführung der protokollierten Operationen zulassen muss. Dies setzt einen aktionskonsistenten Datenbankzustand voraus, welcher ausgeführte Änderungsoperationen entweder vollständig oder gar nicht reflektiert. Da eine Änderungsoperation jedoch in der Regel die Modifikation mehrerer Seiten erfordert, kann die Aktionskonsistenz nur erreicht werden, wenn Änderungen atomar auf den Externspeicher geschrieben werden können.

Die REDO-Recovery erfordert für logisches Logging die Wiederholung vollständiger Datenbankoperationen. Hinzu kommt, dass derselbe Zustand wie im Normalbetrieb nur dann rekonstruiert werden kann, wenn die Operationen im Einbenutzerbetrieb wiederholt werden, was die Recovery-Zeiten zunächst erhöht.

Eine weitere Schwierigkeit logischer Log-Verfahren liegt darin, dass es zur UNDO-Recovery notwendig ist, die Umkehroperation einer durchgeführten Änderung auszuführen. Dies wirft vor allem bei mengenorientierten Änderungen Probleme auf. So erfordert beispielsweise das Löschen aller Mitarbeiter einer Abteilung als UNDO-Operation das Einfügen aller betroffenen Mitarbeiter, was jedoch die Protokollierung aller einzufügenden Attributwerte erfordert. Ähnlich umfangreiche Protokollierungen sind erforderlich, um kaskadierende Löschtätigkeiten zur Wartung der referentiellen Integrität rückgängig machen zu können.

10.2.2 Physisches Logging

Die verbreiteste Form des Logging ist physisches Logging, bei dem die Log-Information auf Ebene physischer Objekte verwaltet wird. Dies sind in heutigen kommerziellen Systemen die Objekte der Speicherungsstrukturen, d.h. die Datensätze und Indexstrukturen. Beim Zustands-Logging werden die Zustände dieser Objekte vor und nach der Änderung im Log-Satz protokolliert, der alte Zustand wird dabei als Before-Image, der neue Zustand als After-Image bezeichnet. Die Recovery besteht dann lediglich darin, diese Werte in die Datenbank zu übernehmen, wobei die Before-Images zur UNDO-Recovery, die After-Images zur REDO-Recovery angewendet werden.

Die Before- und After-Images können beim Übergangs-Logging zu einer Log-Information zusammengefasst werden, indem man diese beiden Images XOR-verknüpft (Zustandsdifferenz) und das Ergebnis in die Log-Datei schreibt. Ist ein REDO durchzuführen, so liegt der alte Wert in der Datenbank. Dieser kann mit dem LOG-Satz wieder XOR-verknüpft werden, wodurch das After-Image entsteht. Analog existiert bei einer notwendigen UNDO-Recovery der After-Imagezustand in der aktuellen Datenbank. Dieser XOR-verknüpft mit dem LOG-Eintrag ergibt das Before-Image.

- ▶ $\text{Log} = \text{BI} \times \text{AI}$
- ▶ $\text{Log} \times \text{AI} = \text{BI} \times \text{AI} \times \text{AI} = \text{BI}$
- ▶ $\text{Log} \times \text{BI} = \text{BI} \times \text{AI} \times \text{BI} = \text{AI}$

Um die Log-Informationen noch kompakter abzulegen, können interne Operationen, die keinen aktionskonsistenten Speicherinhalt erfordern, mit dem physischen Logging kombiniert abgespeichert werden. Dies wird auch als physiologisches Logging bezeichnet.

10.3 Abhängigkeiten zur Pufferverwaltung

Die Realisierung der Logging- und Recovery-Komponente ist sehr stark von der Realisierung anderer Datenbankkomponenten abhängig. Eine wichtige Abhängigkeit existiert zur Pufferverwaltung, insbesondere im Bezug auf das Ausschreiben geänderter Seiten. Dies bestimmt maßgebend den notwendigen Log-Aufwand und die durchzuführenden Maßnahmen im Crash-Recovery-Fall.

10.3.1 Ausschreiben geänderter Seiten

Da nach einem Systemfehler die Hauptspeicherinhalte nicht mehr existieren, wird durch die Ausschreibstrategie bestimmt, welche UNDO- und REDO-Aktionen notwendig sind. Ein UNDO ist dabei nur erforderlich, wenn schmutzige Seiten ausgeschrieben werden, welche Änderungen von noch nicht erfolgreich beendeten Transaktionen enthalten. Umgekehrt ist eine REDO-Recovery nur notwendig, wenn zum Commit-Zeitpunkt einer Transaktion ihre Änderungen noch nicht vollständig ausgeschrieben wurden.

Hinsichtlich des Ausschreibens schmutziger Änderungen bestehen für die Datenbankpufferverwaltung zwei generelle Strategien:

- ▶ NoSteal bedeutet, dass Seiten mit schmutzigen Änderungen nicht aus dem Puffer ersetzt (gestohlen) werden dürfen. Somit ist garantiert, dass die materialisierte Datenbank nach einem Rechnerausfall keine Änderungen von nicht erfolgreich abgeschlossenen Transaktionen enthält. Damit ist keine Undo-Recovery erforderlich.
- ▶ Die Steal-Alternative erlaubt die Ersetzung schmutziger Seiten. Demnach ist nach einem Systemfehler eine UNDO-Recovery erforderlich, um die Änderungen nicht erfolgreicher Transaktionen zurückzusetzen.

Der NoSteal-Ansatz erscheint zwar aus Sicht der Fehlerbehandlung attraktiv, jedoch bewirkt er eine erhebliche Einschränkung der Datenbankpufferverwaltung. Insbesondere für längere Änderungstransaktionen werden große Teile des Puffers blockiert, wodurch sich Nachteile für die Trefferraten ergeben können. In Extremfällen übersteigt die Anzahl geänderter (schmutziger) Seiten die Puffergröße, so dass ein NoSteal-Ansatz hier nicht erreicht werden kann. Die meisten Datenbanksysteme verwenden daher die flexiblere Steal-Variante unter Inkaufnahme einer UNDO-Recovery.

Eine weitere Design-Entscheidung der Datenbankpufferverwaltung betrifft die Frage, ob die Änderungen einer Transaktion bis zum Commit ausgeschrieben sein müssen oder nicht. Dies führt zu folgenden Alternativen:

- ▶ Ein Force-Ansatz verlangt, dass alle geänderten Seiten spätestens zum Transaktionsende (vor dem Commit) in die permanente Datenbank durchgeschrieben werden. Damit entfällt die Notwendigkeit einer REDO-Recovery nach einem Rechnerausfall.
- ▶ Bei NoForce wird dagegen auf das Hinauszwingen der Änderungen verzichtet, stattdessen können die Seiten nach Ende der ändernden Transaktionen geschrieben werden. Dafür ist nach einem Systemausfall eine REDO-Recovery erforderlich, um die noch nicht ausgeschriebenen Änderungen erfolgreicher Transaktionen zu wiederholen.

Trotz des erhöhten Recovery-Aufwands spricht die bessere Leistungsfähigkeit im Normalbetrieb klar für einen NoForce-Ansatz. Denn Force führt zu einem sehr hohen E/A-Aufwand für die Ausschreibvorgänge, da jede Änderung einzeln ausgeschrieben wird. Weiterhin bewirken die Schreibvorgänge eine signifikante Verschlechterung der Antwortzeiten, was aufgrund länger gehaltener Sperrungen auch die Anzahl an Sperrkonflikten erhöht. Mit NoForce entfallen diese Probleme, insbesondere können mehrere Änderungen pro Seite über Transaktionsgrenzen hinweg akkumuliert werden, was insbesondere für größere Puffer deutliche E/A-Einsparungen erlaubt.

Für die beiden Strategien ergeben sich prinzipiell vier Kombinationsformen:

	Steal	NoSteal
Force		
NoForce		

Die Kombination Steal/NoForce erfordert sowohl UNDO- als auch REDO-Recovery, stellt jedoch auch die allgemeinste Lösung mit den im Normalbetrieb im Allgemeinen besten Leistungsmerkmalen dar, welche von den meisten Datenbanksystemen befolgt wird. Eine Kombination NoSteal/Force verspricht dagegen ein Wegfallen von UNDO- als auch REDO-Recovery. Jedoch ist diese Kombination bei nicht atomaren Schreibvorgängen nicht realisierbar. Denn NoSteal verlangt, dass Änderungen einer Transaktion erst nach ihrem Commit in die permanente Datenbank gelangen, was jedoch Force widerspricht, Force verlangt das Ausschreiben von Seiten vor dem Commit.

10.3.2 WAL-Prinzip und Commit-Regel

Damit UNDO- und REDO-Recovery überhaupt korrekt durchgeführt werden können, sind beim Logging zwei fundamentale Regeln zu beachten. Die erste beinhaltet das sogenannte Write-Ahead-Log-Prinzip (WAL-Prinzip). Es besagt, dass vor dem Schreiben einer schmutzigen Änderung in die materialisierte Datenbank die zugehörige UNDO-Information (beispielsweise das Before-Image) in die Log-Datei geschrieben werden muss. Nur dann ist gewährleistet, dass für jede auf der materialisierten Datenbank vorgefundenen schmutzigen Änderung die Log-Information zum Zurücksetzen vorliegt. Die WAL-Regel ist offensichtlich nur für Steal relevant.

Eine analoge Vorschrift, Commit-Regel genannt, besteht für die REDO-Recovery. Sie besagt, dass vor dem Commit einer Transaktion für ihre Änderungen ausreichende REDO-Informationen (beispielsweise After-Images) zu sichern sind. Nur so kann die Wiederholbarkeit einer Transaktion und die Dauerhaftigkeitsgarantie für erfolgreiche Transaktionen sichergestellt werden. Im Fall von NoForce ist die Einhaltung der Commit-Regel Voraussetzung zur Durchführbarkeit der Crash-Recovery. Jedoch auch für Force ist ein Schreiben von REDO-Log-Daten erforderlich, um eine Geräte-Recovery zu ermöglichen.

10.3.3 Commit-Verarbeitung

Die Commit-Verarbeitung folgt dem bereits in Abschnitt 9.4.1 vorgestellten Zwei-Phasen-Commit-Ansatz. Dabei erfolgt das Logging in Phase 1, wobei aufgrund der Commit-Regel zumindest ausreichende Redo-Informationen zu schreiben sind. Im Falle einer Force-Ausschreibstrategie kommt noch das Durchschreiben der geänderten Seiten in die materialisierte Datenbank hinzu, was seinerseits das vorherige Schreiben von UNDO-Informationen auf dem Log erfordert. Phase 1 der Commit-Verarbeitung wird üblicherweise durch Schreiben eines Commit-Satzes auf den Log abgeschlossen, welcher das erfolgreiche Ende der Transaktion feststellt. In Phase 2 erfolgt dann die Freigabe der Sperren, insbesondere für die geänderten Objekte.

Das Ausschreiben der Log-Datei erfolgt dabei aus dem im Hauptspeicher geführten Log-Puffer, in dessen Seiten die Log-Sätze gesammelt werden. Der Log-Puffer kann dabei aus mehreren Seiten bestehen, die in einem Schreibvorgang auf die Log-Platte geschrieben werden. Da die Log-Datei sequentiell beschrieben wird, ergeben sich durch den weitgehenden Wegfall von Zugriffsarmbewegungen relativ schnelle Schreibzeiten. Außerdem lassen sich weit höhere E/A-Raten als bei Platten mit nicht-sequentiellen (wahlfreien) Zugriffen erreichen. Zur Begrenzung der E/A-Anzahl ist die Verwendung kleiner Log-Granulate sehr wichtig, dann können beispielsweise pro Seite 10-20 Änderungen protokolliert werden. Generell erfolgt ein Ausschreiben des Log-Puffers

- ▶ wenn er vollständig gefüllt ist
- ▶ aufgrund der WAL-Regel
- ▶ aufgrund der Commit-Regel

Problematisch ist dabei vor allem die Commit-Regel, die das Ausschreiben des Log-Puffers bei jedem Commit verlangt. Insbesondere für kürzere Transaktionen, deren Log-Daten meist keine

Seite füllen, führt dies zu einer erheblichen Beeinträchtigung des Pufferungseffekts. Eine einfach realisierbare Abhilfe bietet das sogenannte Gruppen-Commit, welches inzwischen von nahezu allen kommerziellen Datenbanksystemen unterstützt wird. Dabei werden die Log-Daten mehrerer Transaktionen im Log-Puffer gebündelt und zusammen ausgeschrieben. Hierbei werden während der Commit-Verarbeitung des Log-Daten zunächst nur in den Log-Puffer eingefügt. Das Ausschreiben des Log-Puffers erfolgt dann im Allgemeinen verzögert, nämlich wenn er vollständig gefüllt ist oder ein Timeout abläuft. Wie praktische Implementierungen gezeigt haben, können mit dieser einfachen Maßnahme die erreichbaren Transaktionsraten signifikant verbessert werden. Werden beispielsweise die Log-Daten von fünf Transaktionen in einer Seite untergebracht, so kann bereits bei einem Log-Puffer von einer Seite die fünffache Transaktionsrate erreicht werden. Bei einem Log-Puffer von mehreren Seiten sind entsprechend höhere Verbesserungen möglich.

Übung 10.1 Füllen Sie nachfolgende Tabelle mit den gegebenen Antwortalternativen aus:

1. Tue überhaupt nichts.
2. Benutze die UNDO-Information und setze zurück.
3. Benutze die REDO-Information und wiederhole.
4. WAL-Prinzip verhindert diese Situation.
5. Zwei-Phase-Commit verhindert diese Situation.

Seite in DB zurückgeschrieben	Log-Satz in Log-Datei	Transaktion nicht beendet	Transaktion abgeschlossen
Nein	Nein		
Nein	Ja		
Ja	Nein		
Ja	Ja		

10.4 Sicherungspunkte

Sicherungspunkte (Checkpoints) stellen Maßnahmen zur Begrenzung des REDO-Aufwands nach Systemfehlern dar. Nach einem Systemfehler ist eine REDO-Recovery für alle erfolgreich geänderten Seiten erforderlich, die zum Fehlerzeitpunkt nur im Datenbankpuffer im Hauptspeicher, jedoch nicht in der materialisierten Datenbank vorlagen. Problematisch sind hier vor allem Hot-Spot-Seiten, welche aufgrund einer hohen Zugriffshäufigkeit nicht zur Verdrängung aus dem Datenbankpuffer ausgewählt werden. Im Extremfall sind für sie alle Änderungen seit dem Start des Datenbanksystems zu wiederholen, was einen sehr hohen REDO-Aufwand bedeuten würde. Dies scheidet im Allgemeinen schon deshalb aus, da zudem ein extremer Platzbedarf für den Log anfallen würde, um alle Log-Sätze seit Start des Datenbanksystems online vorzuhalten. Sicherungspunkte sind somit auch zur Begrenzung des Log-Umfangs erforderlich.

Bei der Realisierung kann zwischen direkten und indirekten Sicherungspunkten unterschieden werden. Direkte Sicherungspunkte verlangen das Einbringen aller geänderten Seiten in die materialisierte Datenbank und damit das Ausschreiben aller geänderten Seiten aus dem Hauptspeicher. Indirekte bzw. unscharfe Sicherungspunkte (fuzzy-checkpoints) umgehen diesen hohen Aufwand, sie protokollieren lediglich gewisse Statusinformationen in der Log-Datei. Hot-Spot-Seiten sind dabei besonders zu behandeln. Während bei direkten Sicherungspunkten eine REDO-Recovery ab dem letzten Checkpoint beginnt, müssen bei einem Fuzzy-Checkpoints auch Änderungen vor dem Checkpoint berücksichtigt werden.

Generell wird die Checkpoint-Durchführung durch spezielle Log-Sätze protokolliert. Da im Falle einer Crash-Recovery der letzte Checkpoint aufgrund der darin enthaltenen Statusinfos entscheidend für den Wiederherstellungablauf ist, wird die Position des letzten Checkpoints innerhalb der Log-Datei in einer separaten Restart-Datei geführt.

Bei der Frequenz von Sicherungspunkten ist ein Kompromiss zwischen Belastung im Normalbetrieb und vertretbarer Dauer der REDO-Recovery zu treffen. Eine seltene Durchführung von Sicherungspunkten verursacht einen hohen REDO-Aufwand, bei häufiger Durchführung entsteht dagegen ein hoher Overhead im Normalbetrieb. Der Abstand zwischen zwei aufeinanderfolgenden Sicherungspunkten kann beispielsweise über eine feste Zeitspanne oder über eine bestimmte Anzahl von seit dem letzten Checkpoint geschriebenen Log-Sätzen festgelegt werden.

10.5 Aufbau der Log-Datei

Die Log-Informationen werden üblicherweise in einer sequentiellen Datei auf der Festplatte gespeichert, wobei die neuen Log-Daten jeweils an das aktuelle Dateiende geschrieben werden. Die Log-Datei muss unabhängig von der permanenten Datenbank auf dedizierten Externspeichern gehalten werden, so dass beim Ausfall der Datenbankplatten die Log-Datei erhalten ist. Zur weiteren Absicherung werden die Log-Dateien in der Regel doppelt geführt (Duplex-Logging), um auch den Ausfall einer Log-Platte verkraften zu können.

10.5.1 Log-Satzarten

Im Log werden u.a. folgende Satzarten unterschieden:

- ▶ Transaktionsbeginn, Transaktions-Commit sowie Transaktions-Rollback, wobei Transaktionen über eine eindeutige Transaktionsnummer identifiziert werden.
- ▶ Log-Sätze zur Beschreibung von Änderungen mit UNDO- und REDO-Informationen.
- ▶ Log-Sätze zur Beschreibung von Sicherungspunkten.

Die Log-Sätze weisen dabei variable Länge auf und haben eine eindeutige Kennung, die Log-Sequence-Number (LSN). Die LSNs werden streng monoton wachsend vergeben, so dass sie die chronologische Reihenfolge der Log-Einträge und damit der protokollierten Änderungen reflektieren. Für physische Logging-Verfahren enthalten die Log-Sätze von Änderungen die Nummer der Seite (PageID), auf die sich die Änderung bezieht. Zudem sind alle Log-Sätze einer Transaktion rückwärts verkettet. Damit wird ein schnelles Transaktions-UNDO unterstützt, bei dem die Änderungen einer Transaktion in umgekehrter zeitlicher Reihenfolge zurückzusetzen sind. Zur Behandlung von Transaktionsfehlern im Normalbetrieb wird für jede laufende Transaktion die LSN des letzten Log-Satzes der Transaktion im Hauptspeicher vermerkt. Die Rückwärtsverkettung umgeht dann einen Scan der Log-Datei.

10.5.2 Begrenzung des Log-Umfangs

Die temporäre Log-Datei dient im Wesentlichen zur Behandlung von Transaktions- und Systemfehlern. Der Umfang der temporären Log-Datei lässt sich dabei im Vergleich zur Datenbankgröße relativ klein halten, da die Log-Daten nur für begrenzte Zeit benötigt werden:

- ▶ Für erfolgreich beendete Transaktionen werden UNDO-Informationen nicht mehr benötigt, da sie nicht mehr zurückgesetzt werden können.

- ▶ Nach dem Ausschreiben einer geänderten Seite in die permanente Datenbank wird die REDO-Information der Änderung für die Crash-Recovery nicht mehr benötigt.¹

Die genannten Kriterien erlauben die Verwendung eines fest begrenzten Log-Umfangs. Üblicherweise wird dabei eine logische Ringpufferorganisation der Log-Datei verwendet, bei der die Einträge der Log-Datei zyklisch überschrieben werden.

10.6 Crash-Recovery

Beim Wiederanlauf (Restart) des Datenbanksystems nach einem Systemfehler ist der jüngste transaktionskonsistente Datenbankzustand herzustellen, der zum Fehlerzeitpunkt gültig war. Hierzu wird die temporäre Log-Datei sowie die materialisierte Datenbank herangezogen. Eine weitere wichtige Forderung ist, dass auch Fehler während des Wiederanlaufs korrekt behandeln werden können. Dies erfordert die Idempotenz der Crash-Recovery, so dass bei mehrfacher Anwendung der REDO- oder UNDO-Informationen jeweils dasselbe Ergebnis geliefert wird.

10.6.1 Überblick zur Restart-Prozedur

Ein verbreiteter Ansatz erfordert die Restart-Verarbeitung innerhalb von drei Phasen, jeweils verbunden mit einem Scan der Log-Datei.

Analyselauf: Zunächst wird beginnend vom letzten Checkpoint-Satz die Log-Datei bis zu ihrem Ende gelesen. Aus der Checkpoint-Information werden die zum Checkpoint-Zeitpunkt laufenden Transaktionen sowie die geänderten Seiten ermittelt. Davon ausgehend wird über den Log die Menge der Gewinner- und Verlierertransaktionen bestimmt. Gewinner sind dabei die Transaktionen, für die ein Commit-Satz im Log gefunden wird. Verlierer dagegen sind Transaktionen, für die ein Rollback-Satz bzw. kein Transaktionsendesatz vorliegt.

REDO-Laufen: In der zweiten Phase erfolgt die Wiederholung der Änderungen, welche noch nicht in den betroffenen Seiten liegen. Der Startpunkt für diese REDO-Recovery ist vom Checkpoint-Typ abhängig und wird bei Fuzzy-Checkpoints auch in die Log-Datei reingeschrieben. Von dort aus wird der Log bis zum Ende gelesen, um die Änderungen in chronologischer Reihenfolge zu wiederholen. Hierzu müssen die betroffenen Seiten zunächst von der materialisierten Datenbank eingelesen werden, falls sie noch nicht aufgrund vorhergehender REDO-Aktionen im Hauptspeicher vorliegen.

UNDO-Laufen: In der dritten Phase erfolgt das Zurücksetzen der Verlierer-Transaktionen. Dazu wird der Log vom aktuellen Ende an rückwärts gelesen, um die Änderungen in umgekehrter Reihenfolge zurückzunehmen. Hierzu müssen die betroffenen Seiten wiederum eingelesen werden, falls sie noch nicht im Hauptspeicher vorliegen. Das Ende der UNDO-Recovery ist durch den Beginn der ältesten Transaktion gegeben, welche zum Zeitpunkt des letzten Checkpoints aktiv war.

Der Aufwand dieser Prozedur ist vergleichsweise hoch, da die Log-Datei dreimal zu lesen ist und alle betroffenen Seiten vom Externspeicher zu holen sind. Der Zeitbedarf für die Recovery lässt sich jedoch über die Häufigkeit von Sicherungspunkten kontrollieren.

¹Längerfristig benötigt wird dagegen die REDO-Information zur Geräte-Recovery, welche daher in einem Archiv-Log geführt wird.

10.6.2 Die PageLSN

Beim physiologischen und physischen Logging bezieht sich jeder Log-Satz einer Änderung auf genau eine Datenbankseite. Dies erlaubt eine einfache Feststellung, ob eine bestimmte Änderung bereits in einer Datenbankseite enthalten ist oder nicht und somit ob eine Wiederholung der Änderung bei einem REDO erforderlich ist. Hierzu wird für jede Seite im Seitenkopf eine PageLSN geführt, welche der LSN des Log-Satzes entspricht, welcher die zuletzt auf der Seite ausgeführte Änderung protokolliert. Die PageLSN stellt somit eine Art Versionsnummer dar, welche sich bei jeder Änderung erhöht.

Eine Änderung, deren Log-Satz eine LSN aufweist, die kleiner oder gleich der PageLSN der entsprechenden Datenbankseite ist, befindet sich somit bereits in der Seite und braucht nicht wiederholt zu werden. Analoges ist für die UNDO-Recovery möglich.

10.7 Geräte-Recovery

Aufgabe der Geräte-Recovery ist die Vermeidung von Datenverlust trotz Fehlern und Ausfällen der Externspeicher, auf denen die materialisierte Datenbank dauerhaft gespeichert ist. Dies betrifft vor allem die Behandlung von Plattenfehlern. Hierzu werden in Datenbanksystemen üblicherweise Archiv-Kopien und Archiv-Log-Dateien verwendet, welche aus Kostengründen meist auf Bandspeicher gehalten werden. Das Zurückgehen auf eine Archivkopie kann aber auch notwendig werden, wenn logische Fehler in den Programmen die Konsistenz des Datenbestands verletzen oder zu einem Datenverlust geführt haben.

Eine Archivkopie stellt einen Schnappschuss der Datenbank bzw. eines Datenbankteils dar, der in periodischen Abständen zu erstellen ist. Alle seit Erstellung einer Archivkopie erfolgten Datenbankänderungen werden im Archiv-Log protokolliert, wobei sich auch hier ein Teil auf Band befinden kann. Die jüngsten Änderungen, die noch nicht im Archiv-Log enthalten sind, werden vom temporären Log ergänzt. Die REDO-Recovery kann wie für die Crash-Recovery durchgeführt werden. Insbesondere lässt sich über die PageLSN entscheiden, ob ein Log-Satz anzuwenden ist.

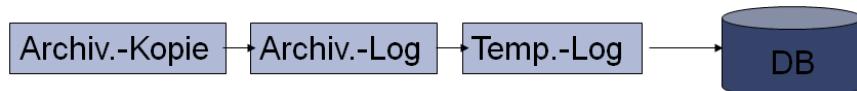


Abbildung 10.4: Datenbankwiederherstellung bei Gerätefehlern

Die Erstellung von Archivkopien oder sogenannte Dumps kann jeweils für die gesamte Datenbank oder für einzelne Datenbankteile erfolgen. Letzterer Ansatz erlaubt die Häufigkeit der Dump-Erstellung an die Änderungsfrequenz des jeweiligen Datenbankbereichs anzupassen. Außerdem wird eine schnelle Recovery ermöglicht, da nur eine Teilmenge der Log-Daten anzuwenden ist.

Der Dump-Prozess kann vollständige oder inkrementelle Archivkopien erzeugen, wobei beides aus Verfügbarkeitsgründen im laufenden Betrieb möglich sein muss. Im ersten Fall wird jede Datenbankseite in der Kopie aufgenommen, während beim inkrementellen Dumping nur die seit der letzten Kopie geänderten Seiten archiviert werden.

10.7.1 Erstellung vollständiger Archivkopien

Die Erstellung vollständiger Archivkopien ist sehr aufwändig. Sie erfordert das Lesen aller Datenbankseiten sowie ihr Ausschreiben in die Archivkopie. Aufgrund der zu bewältigenden Datenmengen können sich dabei Dump-Zeiten im Stundenbereich ergeben.

Die Erstellung einer Archiv-Kopie lässt sich relativ einfach im laufenden Betrieb mit sogenannten Fuzzy Dumps erreichen. Eine solche Archivkopie erfüllt keine Konsistenzanforderungen, sondern

kann Datenbankseiten unterschiedlicher Änderungszustände aufweisen. Zu Beginn des Dump-Prozesses wird die LSN des aktuellen Endes der temporären Log-Datei vermerkt, da alle späteren Änderungen für den Archiv-Log relevant sind. Der eigentliche Dump-Prozess kann als spezielle Lese-Transaktion aufgefasst werden, die alle Datenbankseiten liest und kopiert. Werden keine Lesesperrren gesetzt, kommen auch schmutzige Änderungen in die Archivkopie, die bei der Geräte-Recovery zurückzusetzen sind. Das Setzen kurzer Lesesperrren pro Seite verhindert die Archivierung solcher Änderungen und erlaubt ein vereinfachtes REDO-Recovery.

10.7.2 Alternativen zur Geräte-Recovery

Nachteilig bei der konventionellen Platten-Recovery ist, dass sie manuelle Eingriffe des Systemverwalters erfordert und sehr zeitaufwändig ist. Nach Erkennen eines Gerätefehlers ist die Recovery zu starten, wobei ein zeitaufwändiges Einspielen der Archivkopie sowie eine Anwendung des Archiv-Logs durchzuführen sind, bevor auf die Daten zugegriffen werden kann. Auch wenn Gerätefehler vergleichsweise selten auftreten, ist die damit verbundene Einschränkung der Verfügbarkeit in bestimmten Anwendungsbereichen wie beispielsweise im Bankwesen oder bei Reservierungssystemen, oft nicht akzeptabel. In diesen Fällen kommen folgende Ansätze in Betracht, die eine schnellere Recovery unterstützen:

- ▶ Die Verwendung von Spiegelplatten erlaubt eine sehr schnelle und automatische Behandlung von Plattenfehlern außerhalb des Datenbanksystems auf Hardware oder Betriebssystemebene. Dabei werden alle Daten auf unabhängigen Platten doppelt geführt und beide Kopien stets auf dem aktuellsten Stand gehalten. Nach einem Plattenfehler kann somit die Verarbeitung mit der Kopie ohne Unterbrechung fortgeführt werden. Hauptnachteil dieser Lösung ist die Verdopplung der Speicher Kosten, was vor allem bei sehr großen Datenbanken erheblich zu Buche schlägt. Die Verdopplung der Schreibzugriffe ist weniger gravierend, da diese parallel durchgeführt werden können. Für Lesezugriffe ergeben sich bessere Leistungsmerkmale als bei einfachen Platten.
- ▶ Redundante Disk Arrays (RAID-Systeme) erlauben eine automatische Behandlung von Plattenfehlern zu deutlich geringeren Speichermehrkosten als für Spiegelplatten. Ihre Ausfallsicherheit ist zwar auch sehr groß, jedoch geringer als bei Spiegelplatten. Weiterhin ist nach einem Plattenfehler mit deutlichen Leistungseinbußen zu rechnen, da die vom Ausfall betroffenen Daten zu rekonstruieren sind.
- ▶ Falls eine Kopie der Datenbank in einem geographisch entfernten System geführt wird, kann auch darauf nach Ausfall einer Platte zugegriffen werden. Dieser Ansatz ist zwar vor allem zur Behandlung von Katastrophen von Interesse, um nach vollständigem Ausfall einer Installation eine schnelle Fortsetzung der Datenbankverarbeitung zu erreichen. Die Zusatzkosten sind jedoch sehr hoch, da neben den Externspeichern auch die anderen Systemkomponenten zu duplizieren sind. Zudem verursacht die Aktualisierung der Kopie einen höheren Aufwand im Normalbetrieb als bei beiden zuvor genannten Ansätzen.

Auch bei Verwendung dieser Ansätze werden in der Regel weiterhin Archivkopien und Archiv-Logs benötigt. Denn obwohl ein Datenverlust nur beim Auftreten von Doppelfehlern möglich ist und diese äußerst unwahrscheinlich sind, ist ihr Auftreten vor allem bei einer großen Anzahl von Platten und Systemkomponenten nicht völlig auszuschließen. Außerdem kann es auch ohne Vorliegen von Externspeicherfehlern in bestimmten Situationen notwendig werden, auf ältere Datenbankzustände zurückzugehen. Gründe können ein versehentliches Löschen einer Tabelle oder ein Virusbefall sein, der den Datenbestand löscht.

Kapitel 11

Physische Datenorganisation

Datenbanksysteme werden entwickelt um potentiell sehr große Mengen an Daten zu verwalten, für die Anwendungen abstrakte Sichten auf die Daten bereitzustellen und auf ihre Anforderung hin standardisierte, komplexe Operationen auf den Daten effizient und zuverlässig auszuführen. Ideal wäre hierfür ein Speicher mit nahezu unbegrenzter Speicherkapazität, kurzer Zugriffszeit bei wahlfreiem Zugriff, hohen Zugriffsraten und geringen Speicherkosten. Außerdem müsste er nicht-flüchtig sein, um Objekte persistent halten zu können. Diese Idealvorstellung würde es gestatten, alle vernünftigen Leistungsansprüche, die an die DB-Verarbeitung gestellt werden, zu befriedigen. Solche Speicher gibt es jedoch nicht, so dass das gewünschte Systemverhalten durch Nutzung einer passenden Hardware-Architektur angenähert werden muss.

11.1 Einsatz von Speichermedien

Sowohl Prozessor als auch Externspeicheranbindung spielen eine dominierende Rolle für das Leistungsverhalten eines DBS. Deshalb sind Systemengpässe durch gezielte Maßnahmen zur Geschwindigkeitsanpassung zu entschärfen oder durch Verlagerung der Verarbeitung zu entlasten. Gerade der Externspeichereinsatz sollte gut geplant werden, um Engpässe frühzeitig zu verhindern.

Große Datenbanken können nicht mehr auf eine Festplatte abgespeichert werden, der Datenbestand muss auf mehrere Platten verteilt werden. Zur möglichst optimalen parallelen Nutzung der Platten und einer damit verbundenen Lastverteilung ist eine sinnvolle Verteilung der Daten (Declustering) von großer Bedeutung. Voraussetzung hierfür ist, dass man die Größe der Tabellen und die Zugriffshäufigkeit auf die einzelnen Tabellen genau im Voraus abschätzen kann. Ist dies bekannt, kann man Tabellen in Teile aufspalten und diese je nach Zugriffshäufigkeit auf die Platten verteilen. Dieser Vorgang wird auch partitionieren genannt und findet sich beispielsweise bei ORACLE innerhalb der PARTITION-Klausel des CREATE TABLE-Befehls. So kann man Tabellen und Indizes in relationalen DBS über mehrere Platten aufgrund der Werte eines Attributs verteilen. Dadurch wird eine effektive Parallelisierung von Operationen unterstützt, so dass parallele Teiloperationen auf verschiedenen Platten arbeiten können.

Alternativ können die Daten in Einheiten einer Blockgröße vom Plattencontroller auf mehrere Platten verteilt werden. Aufgrund der sehr kleinen Verteilungsgranularität wird hierdurch die Last sehr gleichmäßig auf die Platten verteilt. Auch sind diese Lösungen typischerweise im Verbund mit RAID-Lösungen und damit ausfallicherem Platteneinsatz zu finden.

11.2 Dateien

Die physischen Datenobjekte eines DBS werden auf nichtflüchtigen Externspeichern während ihrer gesamten Lebenszeit aufbewahrt. Ein geeignetes Dateikonzept verdeckt die physischen Aspekte und technologieabhängigen Charakteristika der Externspeicher. Es wird durch ein Dateisystem realisiert, das eine Menge von Dateien verwaltet und auch alle Zugriffsoperationen auf ihnen abwickelt. Dateien repräsentieren externe Speichermedien in einer geräteunabhängigen Weise und bieten den zugreifenden Programmen eine abstrakte Sicht für ihre Verarbeitungslogik. Im einfachsten Fall lassen sich Dateien als große Daten-Container oder als Menge durchnummierter Blöcke auffassen, wobei Lese- und Schreibzugriffe über die Blocknummer erfolgen können.

Obwohl prinzipiell für die physischen Blöcke der Datei variable Längen denkbar sind, werden konstante und gleichförmige Blocklängen verwendet. Sie gestatten neben der einfachen Adressierung die flexible Ausnutzung des Speicherplatzes ohne Fragmentierungsprobleme. Weiterhin wird dadurch die Pufferverwaltung vereinfacht und eine saubere Schnittstelle für die Geräteunabhängigkeit definiert. Übliche Blockgrößen liegen zwischen 2 und 32 KB.

11.2.1 Tablespaces

Im Rahmen der Dateierzeugung zur Speicherung der Tabellendaten werden zwei prinzipielle Vorgehensweisen unterschieden. Zum einen kann für jede Relation (und jeden Index) der Datenbank eine separate Datei erzeugt werden. Diese Datei wird vom Betriebssystem verwaltet, die Größe der Datei passt sich immer der Größe der Tabelle an. Werden neue Datensätze in die Datei geschrieben, wird dadurch, sofern noch Platz auf der Platte ist, die Datei vergrößert. Dies ist insbesondere für Tabellen interessant, deren Größe im Vorfeld nicht abgeschätzt werden kann, die jedoch auch nicht zu groß werden können.

Tablespaces (Tabellenbereiche) dagegen bestehen aus Dateien fester Größe. In einem Tablespace können beliebig viele Datenbanktabellen und andere Datenbankinhalte gespeichert werden. Da die Verwaltung der Datei dem Datenbanksystem überlassen wird, kann die Verwaltung optimal auf die Anforderungen der Datenbank eingestellt werden. Deshalb ist die Tablespace-Lösung meist schneller.¹ Abbildung 11.1 veranschaulicht etwas von der Flexibilität, die man durch die Verteilung von Daten auf Tabellenbereiche erhält. Ein Tablespace kann über eine oder mehrere physische Speichereinheiten verteilt angelegt werden.

11.2.2 Externe Dateien

Dateisysteme verfügen in der Regel nicht über genügend Metadaten, um ihre Dateioperationen durch Suchfunktionen als auch durch Maßnahmen zur Integritätskontrolle anreichern zu können. Andererseits sind DBS nicht auf die kosteneffektive Speicherung und Verwaltung einer großen Anzahl von BLOBs (Binary Large OBjects) eingerichtet, etwa um Bibliotheken von Multimedia-Objekten in integrierter Weise zu unterstützen. Derartige Objekte benötigen eine hierarchische Speicherorganisation mit Hilfe von Tertiärspeichern, die auf die Verwaltung dieser Daten und ihrer variierenden Zugriffsmuster zugeschnitten sind. Deshalb bietet es sich an, bestimmte Konzepte von DBS wie Sicherung der referentiellen Integrität, Zugriffskontrolle, Backup und Recovery mit den Vorzügen der Dateisysteme bei der Speicherung und dem spezialisierten Zugriff auf BLOBs zu kombinieren. Voraussetzung ist die Einrichtung von speziellen Dateisystemerweiterungen, die eine Zusammenarbeit mit dem DBS über vordefinierte Protokolle erlauben. Bei einem solchen Ansatz übernimmt das DBS die Rolle und Funktion eines Metadaten-Repository, in dem mit Hilfe von Zeigern auf die in Dateisystemen verbleibenden Daten verwiesen wird.

¹IBM DB2 unterstützt beide Varianten der Dateierzeugung. Beide werden hier Tablespace genannt, es wird dabei zwischen systemverwalteten und datenbankverwalteten gesprochen. Es gibt auch Datenbanksysteme, die das Betriebssystem hier vollständig umgehen und direkt eine Festplattenpartition verwalten. Dies bietet eine optimale Performance, da hierdurch die Betriebssystemebene ausgeblendet wird.

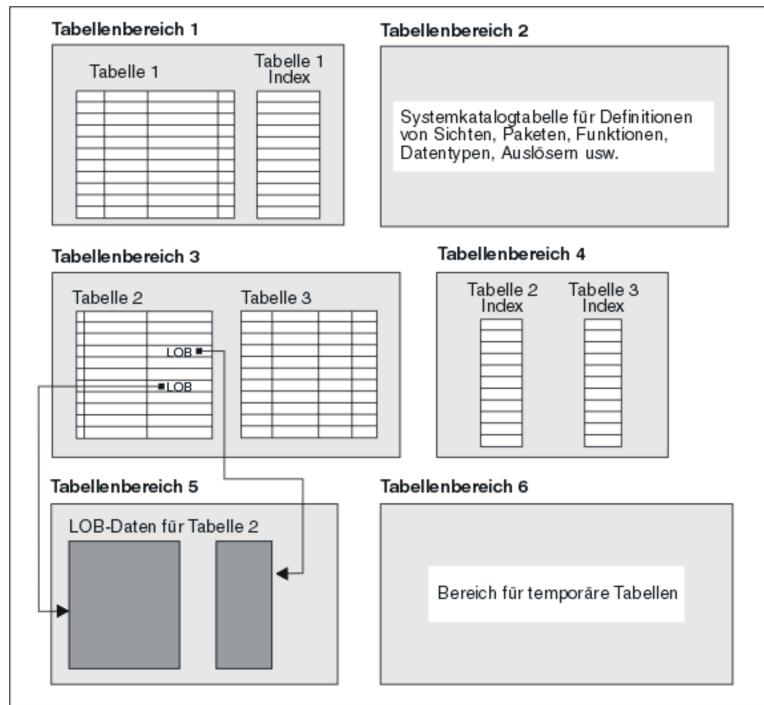


Abbildung 11.1: Verschiedene Tablespace

In Abbildung 11.2 ist der Ansatz skizziert, der im DataLinks-Projekt verfolgt wird. In einer Relation werden Beschreibungsdaten für die externen Dateien gespeichert, die in großer Anzahl in verschiedenen weltweit verteilten Datei-Servern gehalten werden können. Um eine inhaltsorientierte Suche über die externe Datei-Menge durchführen zu können, sind Beschreibungsinformationen in der Datenbank abzulegen. Für Referenzen wurde ein spezieller EFR-Datentyp (External File Reference) eingeführt. Er ist auf dem URL-Konzept (Uniform-Resource Locators mit Dateiname und Serverangabe) aufgebaut, das eine weltweite Verteilung über heterogene Rechnernetze ermöglicht. Für die in einer SQL-Relation verwalteten URLs erzwingt das DBS in Zusammenarbeit mit den die Datenquellen verwaltenden Dateisystemen die Einhaltung der referentiellen Integrität, die URL-Adressen sind stets gültig und die referenzierten Dateien können nicht unkontrolliert gelöscht oder verschoben werden.

Zur Verbesserung der Sicherungsmaßnahmen und der Datenqualität lassen sich als weitere DBS-Funktionen Backup und Recovery für die externen Dateien nutzen. Unter der Annahme, dass die über URLs referenzierten Dateien weitgehend unverändert bleiben, ist eine Koordination der Durchführung von Backups (für Archivkopien) und Recovery von DB-Daten und von den über URLs referenzierten Dateien in den verschiedenen Datei-Servern möglich und sinnvoll, um ein gemeinsames konsistentes Zurücksetzen oder eine Wiederherstellung eines Datenzustands zu bewerkstelligen. Dazu sind folgende Absprachen erforderlich:

- Wenn ein DB-Backup durchgeführt wird, müssen alle Datei-Backups beendet sein, bevor der DB-Backup als erfolgreich erklärt wird.
- Wenn bei einem DB-Restore Referenzen zu Dateien betroffen sind, werden die betreffenden Dateien ebenfalls zurückgesetzt.

Die Umsetzung dieser Ideen wird mit folgender Systemkonfiguration (Abbildung 11.3) realisiert: Die Anwendung kann über SQL beim DBS die URL erfragen. Zusammen mit den ausgehändigten Token zur Autorisierung wendet sie sich dann direkt an die betreffenden Datei-Server, die

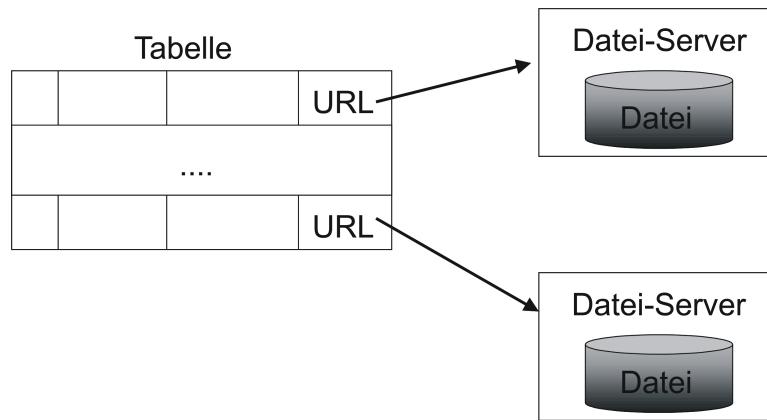


Abbildung 11.2: DBS-Kontrolle bei externen Datenquellen

mit DLFM (DataLinks File Manager) und DLFF (DataLinks Filesystem Filter) spezielle, für die Datalinks-Protokolle vorbereitete Komponenten besitzen. Über einen Kontrollpfad werden Entscheidungen und Zugriffsbeschränkungen zwischen DBS und Datei-Server direkt ausgehandelt.

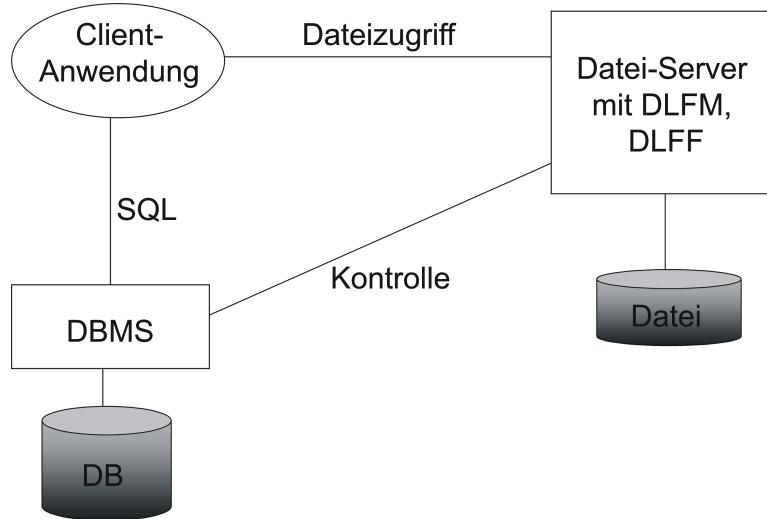


Abbildung 11.3: Systemkonfiguration für Datalinks

11.3 DB-Pufferverwaltung

Datenbankserver verfügen über mehrere GByte an Hauptspeicher, die zur Pufferung von Datenbankinformationen genutzt werden können, wodurch die Zugriffshäufigkeit auf die Plattenlaufwerke reduziert werden kann. Nach Anforderung von einem Client werden die benötigten DB-Objekte in diesen Puffer zum Lesen oder Ändern in Einheiten von Seiten bereitgestellt. Dort ist ihre direkte Adressierung und Manipulation möglich. Falls ein angefordertes DB-Objekt nicht schon im Puffer steht, muss, sofern Speichermangel herrscht, durch Ersetzung freier Platz geschaffen werden. Wurde das zu ersetzende DB-Objekt verändert, ist bei der Externspeicherverwaltung sein Rückschreiben in die Datenbank zu veranlassen, bevor das angeforderte DB-Objekt eingelesen werden kann.

11.3.1 Unterschiede zur BS-Speicherverwaltung

Zur Verwaltung von seitenorientierten virtuellen Speichern und zur Optimierung der Dateiverarbeitung sind heutige Betriebssysteme mit einer ähnlichen Funktion ausgestattet. Sie bieten Pufferbereiche im Hauptspeicher, durch die eine größtmögliche Nutzung der Referenzlokalität bei den Zugriffen der Anwendungen und entsprechend eine Minimierung der E/A-Operationen erzielt werden soll. Dies kann aber aus folgenden Gründen für Datenbanksysteme nicht eingesetzt werden:

- ▶ Betriebssysteme unterstützen DB-spezifische Referenzmuster nicht. Die Ersetzungsverfahren sind auch auf zyklisch sequentielle oder baumartige Zugriffsfolgen abzustimmen.
- ▶ Prefetching von Seiten ist für das Leistungsverhalten von sequentiellen DB-Operationen sehr wichtig, da sich aufgrund von Seiteninhalten oder Referenzmustern oft eine Voraussage des Referenzverhaltens machen lässt.
- ▶ Ein Schreiben in eine Seite führt bei der BS-Pufferverwaltung normalerweise nicht direkt zu einer Ausgabe auf den Externspeicher. Ein selektives und gezieltes Ausschreiben von Seiten zu bestimmten Zeitpunkten, was für das Logging in DBS unbedingt notwendig ist, ist deshalb nicht möglich.

Aus diesen Gründen muss ein DBS in seinem Adressraum seine eigene Pufferverwaltung realisieren. Da keine Hardware-Unterstützung für Benutzerprozesse geboten wird, sind alle ihre Funktionen in Software zu implementieren. Dadurch ist die erforderliche Flexibilität möglich, mit der sich die skizzierten Probleme beseitigen lassen.

In einem DBS sind oft eine Vielzahl von unterschiedlich großen Arbeits- und Pufferbereichen, zu verwalten. Da überall ähnliche Funktionen anfallen, beschränken sich die folgenden Ausführungen auf den DB-Puffer, über den primär die Lese- und Schreibvorgänge aller parallelen Transaktionen abgewickelt werden. Er besteht aus p im Hauptspeicher angeordneten Pufferrahmen, in denen zu jedem Zeitpunkt bis zu p gleichgroße DB-Seiten temporär zwischengespeichert sein können. Da alle parallelen Transaktionen um die verfügbaren Pufferrahmen konkurrieren, sind bei der Verwaltung des DB-Puffers zur optimalen Abwicklung der Anforderungen eine Reihe von Kontroll- und Zuteilungsaufgaben zu erfüllen.

Für die Leistungsfähigkeit des gesamten DBS sind die Größe und die Verwaltung des DB-Puffers durch geeignete Such- und Ersetzungsalgorithmen von entscheidender Bedeutung. Die erforderliche Größe des DB-Puffers hängt in hohem Maße von der Art und dem Grad der Parallelität der beabsichtigten Anwendungen ab. Sie lässt sich als Parameter zur Startzeit des DBS festlegen. Die Größe bleibt jedoch meist kleiner als die Datenbank, so dass der Puffer immer nur einen Bruchteil der Datenbankseiten aufnehmen kann. Bei den Datenbankzugriffen ist ein hohes Maß an Referenzlokalität erkennbar, gerade referenzierte Seiten besitzen eine deutlich erhöhte Wiederbenutzungswahrscheinlichkeit. Physische E/A-Vorgänge können dann reduziert werden, wenn versucht wird, häufig benutzte Datenbankseiten in DB-Puffer zu halten. Erst wenn es gelingt, die Fehlseitenrate auf unter 5% zu senken, wird der leistungshemmende Einfluss der Zugriffslücke auf die DB-Verarbeitung spürbar nachlassen.

11.3.2 Allgemeine Arbeitsweise

Wird zur Bearbeitung einer SQL-Anweisung ein Datensatz aus einer Seite benötigt, muss diese zur Befehlausführung im Hauptspeicher zur Verfügung stehen. Diese Situation wird als Seitenreferenz bezeichnet. Bei der Bereitstellung durch die DB-Pufferverwaltung lassen sich folgende Fälle unterscheiden:

- ▶ Die benötigte Seite ist bereits im DB-Puffer vorhanden. Für die Pufferverwaltung ist ein Aufwand von ca. 100 Instruktionen erforderlich.

- Die benötigte DB-Seite ist nicht im DB-Puffer. Nach der erfolglosen Suche im DB-Puffer muss sie durch eine physische E/A-Operation über die Speicherverwaltung vom Externspeicher geholt werden. Da in der Regel kein Rahmen für eine neue Seite in DB-Puffer frei ist, hat der Pufferverwalter vorher eine im Puffer befindliche Seite zum Ersetzen auszuwählen. Wurde diese Seite verändert, muss sie auf den Externspeicher zurückgeschrieben werden bevor die neue Seite gelesen und bereitgestellt werden kann. Dabei fällt für jede physische E/A-Operation ein CPU-Aufwand von 2500 Instruktionen und ein durch den Externspeicher vorgegebener Aufwand von 10-20 ms an.

11.3.3 Ersetzungsverfahren für Seiten

Falls eine logische Seitenreferenz im DB-Puffer nicht befriedigt werden kann, muss eine Seite zur Ersetzung ausgewählt werden. Die auszulagernde Seite wird dabei durch das Ersetzungsverfahren bestimmt. Seitenersetzungsverfahren lassen sich einteilen in Prefetching- und Demand-Fetching-Verfahren.

11.3.3.1 Prefetching

Durch Prefetching sollen Seiten, die demnächst mit hoher Wahrscheinlichkeit referenziert werden, schon vorab in den DB-Puffer gebracht werden, um synchrone E/A-Wartezeiten soweit wie möglich zu vermeiden. Die zu holende Seitenmenge und das Auslösen des Ersetzungsvorgangs können hierbei unabhängig gewählt werden. Die Tauglichkeit von Prefetching-Verfahren hängt vor allem von der Güte der Vorplanung und Abschätzung künftiger Referenzen ab, da im voraus eingelesene, aber nicht referenzierte Seiten nur unnötigen Leseaufwand verursachen und vielleicht Seiten verdrängen, die demnächst gebraucht werden. Der Einsatz von Prefetching bietet sich für sequentielle Verarbeitungsoperationen (Durchsuchen einer vollständigen Tabelle) an, deren Abwicklung sich dadurch erheblich beschleunigen lässt. So gestattet beispielsweise ORACLE für sequentielle Scans die Einrichtung spezieller Wechselpuffer, in die durch Prefetching abwechselnd Seiten geladen werden können, so dass das Lesen und Durchsuchen von Seiten hochgradig überlappend erfolgen kann.

Physisch sequentielle Lesevorgänge können durch fortlaufende Seitenanforderungen sogar automatisch von der DB-Pufferverwaltung erkannt werden. Jedoch lassen sich auch logisch sequentielle Operationen unterstützen, wenn der DB-Pufferverwaltung geeignete Hinweise zur Verfügung gestellt werden. Der größte Effizienzgewinn wird beim Prefetching von physisch benachbarten Seiten erreicht, da sich hierbei das im Vergleich zu entsprechend vielen wahlfreien Zugriffen wesentlich günstigere Zugriffsverhalten ausnutzen lässt.

11.3.3.2 Demand-Paging

Die am häufigsten eingesetzte Verfahrensklasse ist das Demand-Paging, bei dem bei Auftreten einer Fehlseitenbedingung die angeforderte Seite vom Externspeicher geholt wird, nachdem für sie im DB-Puffer Platz geschaffen wurde. Verfahren zur Auswahl der zu verdrängenden Seite ersetzen diejenige Seite im DB-Puffer, deren Erwartungswert für ihre Wiederbenutzung minimal ist.

Sie müssen dazu Kenntnisse des bisherigen Referenzverhaltens ausnutzen, um das zukünftige Verhalten zu extrapolieren. Dabei wird im Allgemeinen wegen der in Referenzfolgen beobachteten Lokalität davon ausgegangen, dass das jüngste Referenzverhalten ein guter Indikator für die nähere Zukunft ist. Als Bestimmungskriterien dafür eignen sich vor allem das Alter und die Referenzen einer Seite im DB-Puffer. Da die meisten Verfahren ein oder beide Kriterien bei ihrer Ersetzungsentscheidung heranziehen, lassen sie sich vorteilhaft zu ihrer Klassifikation benutzen. Dabei ist es nützlich zu unterscheiden,

- ob das Alter seit Einlagerung, seit letztem Referenzzeitpunkt oder überhaupt nicht und

- ob alle Referenzen, die letzte Referenz oder keine

bei der Auswahlentscheidung des Verfahrens zum Tragen kommen.

FIFO (First-In First-Out) ersetzt diejenige Seite, die am längsten im Puffer ist. Unabhängig von der Referenzhäufigkeit entscheidet allein das Alter einer Seite seit der Einlagerung. FIFO eignet sich deshalb nur bei strikt sequentiellen Zugriffsverhalten, da Seiten, die häufig benötigt werden, bei diesem Verfahren genau so verdrängt werden, wie Seiten mit einmaliger Nutzung.

LFU (Least Frequently Used) konzentriert sich ausschließlich auf das zweite Ersetzungskriterium. Es ersetzt die Seite im Puffer mit der geringsten Referenzhäufigkeit. Dazu müssen die Häufigkeiten der Seitenreferenzen explizit aufgezeichnet werden. Bei einer Ersetzungsanforderung wird die Seite mit dem kleinsten Wert des Referenzzählers ausgelöst, wobei bei einer Pattsituation frei zwischen den beteiligten Seiten gewählt wird. Das Alter einer Seite spielt bei einer solchen strikten LFU-Realisierung keine Rolle, sondern ausschließlich die Häufigkeit ihrer Wiederbenutzung, so dass Seiten, die einmal kurzzeitig außerordentlich häufig referenziert wurden, praktisch nicht mehr zu verdrängen sind, selbst wenn sie später nie mehr angefordert werden. Aus diesem Grund verbietet sich die Implementierung von LFU. Durch zusätzliche Maßnahmen, beispielsweise periodisches Herabsetzen der Referenzzähler (Altern), erhöht sich seine Tauglichkeit, es verliert jedoch auch seine ursprünglichen Charakteristika.

Weitere Verfahren berücksichtigen sowohl Alter als auch Referenzen. Ein beliebtes Verfahren (beispielsweise System R) ist LRU (Least Recently Used), das diejenige Seite im Puffer ersetzt, die am längsten nicht mehr angesprochen wurde. Die Ersetzungsentscheidung wird also allein durch das Alter seit der letzten Benutzung und damit auch durch die letzte Referenz bestimmt. LRU ist jedoch ungeeignet, wenn häufige sequentielle Zugriffe wie ein kompletter Tabellendurchlauf auftreten. Hierdurch wird der DB-Puffer vollständig ausgetauscht und die Referenzlokalität der Anwendungssituation geht verloren, da Referenzhäufigkeiten unberücksichtigt bleiben.

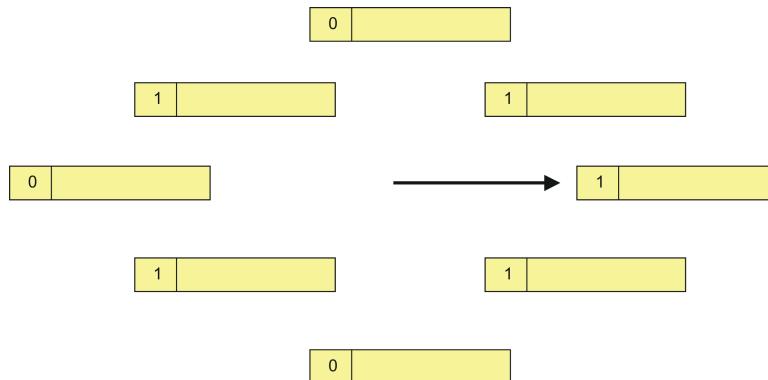


Abbildung 11.4: Clock-Algorithmus

Der Clock-Algorithmus versucht, das LRU-Verhalten mit Hilfe einer einfacheren Implementierung zu erreichen. Er kann anschaulich durch Abbildung 11.4 dargestellt werden. Jede Seite besitzt ein Benutzt-Bit, das bei jeder Seitenreferenz auf 1 gesetzt wird. Bei einer Fehlseitenbedingung wird eine zyklische Suche über die Seiten mit Hilfe eines Auswahlzeigers gestartet, wobei das Benutzt-Bit jeder Seite überprüft wird. Falls es auf 1 steht, wird es auf 0 gesetzt und der Auswahlzeiger wandert zur nächsten Seite. Die erste Seite, deren Benutzt-Bit auf 0 steht, wird zur Ersetzung ausgewählt. Bei dieser Implementierungsform überlebt jede Seite mindestens einen Zeigerumlauf. Clock wurde nicht als Konzept, sondern direkt als Implementierungsform eines Ersetzungsverfahrens entwickelt. Deshalb lassen sich seine Ersetzungskriterien nur annähernd mit Alter seit dem letzten Referenzzeitpunkt und letzte Referenz (wie LRU) klassifizieren. In den erzielten Fehlseitenraten stimmt Clock oft mit LRU überein, hat damit auch dieselben Nachteile bei sequentiellen Tabellendurchläufen, da Referenzhäufigkeiten unberücksichtigt bleiben.

Im Gegensatz zu Ersetzungsverfahren in Betriebssystemen, die oft durch Hardware-Einrichtungen direkt unterstützt werden, müssen solche in DBVS in allen Einzelheiten softwaretechnisch realisiert werden. Diese Notwendigkeit bietet jedoch größere Freiheitsgrade bei der Wahl und Verknüpfung von Ersetzungskriterien. Im folgenden werden solche komplexeren, in DBVS einsetzbaren Ersetzungsstrategien diskutiert.

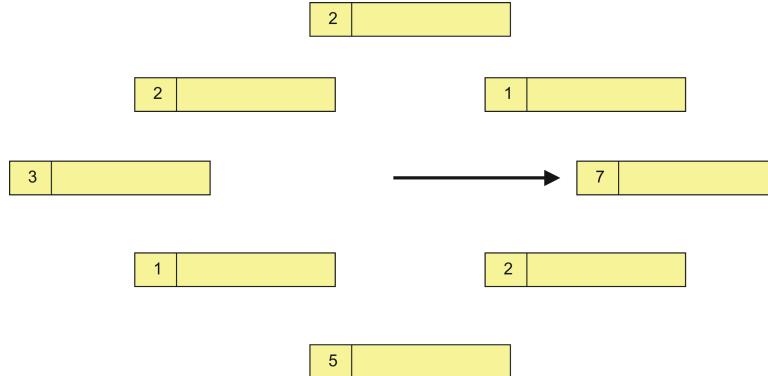


Abbildung 11.5: GClock-AlGORITHMUS

Durch die Kombination der Grundidee von LFU und der Implementierungsform von Clock entsteht der GClock-Algorithmus (Generalized Clock). Das Benutzt-Bit wird durch einen Referenzähler (RZ) ersetzt. Bei einer Fehlseitenbedingung wird eine zyklische Suche mit schrittweisem Herabsetzen der Referenzähler begonnen. Kommt der Zeiger bei der Suche zu einer Seite mit einem Wert ungleich 0 wird jetzt der Wert des Zählers verringert. Dies läuft so lange, bis der erste Referenzähler mit dem Wert 0 gefunden wird. Durch diese Vorgehensweise ergibt sich eine wesentliche Verbesserung im Vergleich zur reinen LFU-Strategie. Trotzdem tendiert GClock dazu, häufig die jüngsten Seiten zu ersetzen, unabhängig von ihrer Art und ihrer tatsächlichen Wiederbenutzungswahrscheinlichkeit. Zur Verbesserung dieses unerwünschten Verhaltens besitzt GClock eine Reihe von Freiheitsgraden:

- ▶ Die Initialisierung des Zählers bei Erstreferenz.
- ▶ Die Inkrementierung des Zählers bei jeder weiteren Referenz.
- ▶ Die Möglichkeit, seitentyp- oder seitenspezifische Maßnahmen zu treffen.

GClock stellt damit eine Klasse von Verfahren dar, in der die einzelnen Varianten durch geeignete Wahl der Parameter auf spezielle Anwendungen hin optimiert werden können.

Bei den bisherigen Verfahren (außer FIFO) wirkte sich das Alter einer Seite höchstens indirekt (über den letzten Referenzzeitpunkt) aus. Es erscheint erfolgsversprechend, Alter und Referenzhäufigkeit direkt in Beziehung zu bringen und als Referenzdichte bei der Ersetzungsentcheidung unmittelbar zu berücksichtigen. Diese Idee lässt sich durch folgenden Algorithmus (in der Grundversion) realisieren: Ein globaler Zähler (GZ) enthält zu jedem Zeitpunkt die Anzahl aller logischen Referenzen. Für jede Seite wird die Nummer ihrer Erstreferenz (EZ) gespeichert, die ihrem Einlagerungszeitpunkt entspricht. Außerdem wird in einem Referenzähler (RZ) die Häufigkeit ihrer Benutzung festgehalten. Somit kann zu jedem Zeitpunkt die Zeitspanne, in der eine Seite ihre Referenzen sammelt, und damit ihre mittlere Referenzdichte (RD) bestimmt werden. Die Referenzdichte für eine Seite S lässt sich mit der Formel $RD_S = RZ_S / (GZ - EZ_S)$ berechnen. Bei einer Fehlseitenbedingung wird die Seite mit der kleinsten mittleren Referenzdichte ausgewählt. Dieses Verfahren wird LRD-V1 (Least Reference Density) genannt.

Bei LRD-V1 unterstellt das Berechnungsverfahren für die Auswahlentscheidung, dass alle Referenzen gleichmäßig seit dem Einlagerungszeitpunkt angefallen sind. Referenzen, die kurzzeitig nach

dem Einlagerungszeitpunkt gehäuft auftreten, halten eine Seite möglicherweise ungerechtfertigt lange im DB-Puffer, da nur die Häufigkeit der Referenzen, aber nicht deren aktuelle Verteilung zur Auswahlentscheidung genutzt werden.

Das Ziel, das Gewicht früherer Referenzen, besonders wenn sie gehäuft aufgetreten sind, bei der Auswahlentscheidung umso deutlicher zu reduzieren, je älter sie sind, lässt sich durch die LRD-V2-Variante erreichen. Nach jeweils festen Referenzintervallen geeigneter Größe, die sich über den globalen Referenzzählern GZ feststellen lassen, werden die einzelnen Referenzzählern durch Subtraktion einer Konstanten bzw. Rücksetzen auf einen konstanten Wert oder durch Division herabgesetzt, so dass eine Art periodisches Altern eintritt. Durch diese Vorgehensweise bestimmen die Referenzen im aktuellen Intervall die Referenzdichte in stärkerem Maße als die in früheren Intervallen. In Tabelle 11.1 sind die beschriebenen Ersetzungsverfahren nach der Art von Berücksichtigung von Alter und Referenzen klassifiziert.

	Alter unberücksichtigt	Alter seit letzter Referenz	Alter seit Einlagerung
Keine Referenzen	Random		FIFO
Letzte Referenz		LRU, Clock,	
Alle Referenzen	LFU	GClock	LRD

Tabelle 11.1: Klassifikation der Ersetzungsverfahren

11.4 Satzverwaltung

Die Speichereinheit auf Platte und die Verwaltungseinheit im DB-Puffer ist die Seite. Jede Seite beginnt dabei mit einem Seitenkopf, der allgemeine Informationen zu der Seite wie eine eindeutige ID, den Seitentyp sowie Informationen zum freien Speicherplatz innerhalb der Seite enthält. Dies umfasst beispielsweise bei IBM DB2 76 Byte. Danach folgen die in der Seite gespeicherten Datensätze.

11.4.1 Externspeicherbasierte Adressierung

Die Datensätze innerhalb einer Seite können von anderen Elementen der Datenbank wie Zugriffstrukturen oder anderen Datensätzen referenziert werden. Deshalb ist es erforderlich, Verweise auf einen Datensatz in der Datenbank abzulegen, die ein schnelles Auffinden des referenzierten Satzes ermöglichen. Dabei ergeben sich folgende Anforderungen:

- ▶ Schneller, möglichst direkter Zugriff auf einen Datensatz.
- ▶ Hinreichend stabil gegen Verschiebungen des Satzes, damit eine geringfügige Verschiebung keine Lawine an Folgeänderungen in den Zugriffspfaden auslöst.
- ▶ Vermeidung zu häufiger Reorganisationen.

Die direkte Adressierung mit Hilfe der relativen Byteadresse (gerechnet vom Datei- oder Seitenanfang) unterstützt zwar den schnellen Zugriff, ist aber nur tragbar, wenn Datensätze nie verschoben werden. Deshalb ist bei der Satzadressierung eine Form der Indirektion vorzusehen, die Verschiebungen innerhalb einer Seite ohne Rückwirkungen zulässt, aber nach Möglichkeit keine weiteren Zugriffskosten einführt.

Das sogenannte TID-Konzept (Tuple-Identifier) dient zur Adressierung innerhalb einer Datei. Es weist jedem Satz ein TID zu, bestehend aus Seitennummer (3-4 Byte) und Index (1-2 Byte) zu einer seiteninternen Tabelle. Die relative Position des Satzes innerhalb der Seite findet sich in dem

durch den Index beschriebenen Tabelleneintrag. Auf diese Weise können Sätze, durch Wachstum oder Schrumpfung (auch anderer Sätze) bedingt, innerhalb der Seite verschoben werden, ohne dass das TID als extern sichtbare Adresse zu ändern ist. Wenn ein Satz aus seiner Hauseite bei starkem Anwachsen oder bei einer Neueinteilung des Speicherplatzes ausgelagert werden muss, kann sein TID durch eine Überlauftechnik stabil gehalten werden. In der Hauseite wird anstelle des Satzes ein Stellvertreter-TID gespeichert, das in der Art eines normalen TID auf den Satz in der neu zugewiesenen Seite zeigt.

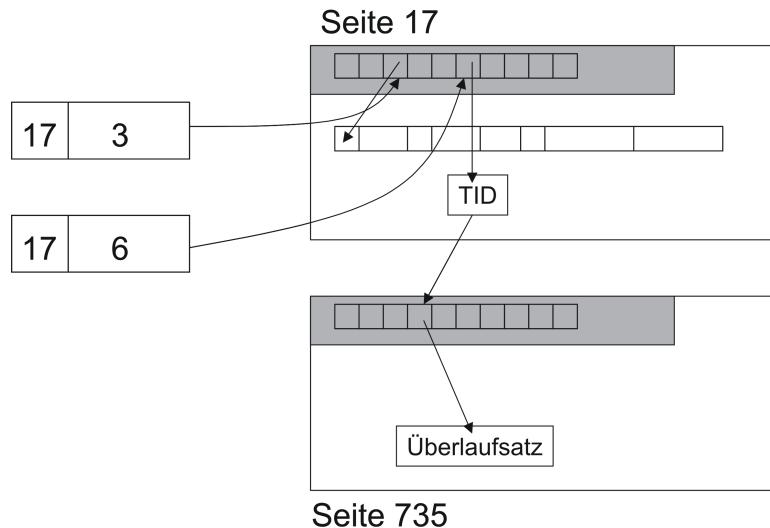


Abbildung 11.6: TID-Konzept

In Abbildung 11.6 ist die Wirkungsweise des TID-Konzepts skizziert. Durch die Regel, dass ein Überlaufsatz nicht wieder überlaufen darf, wird die Überlaufkette auf die Länge 1 beschränkt. Falls ein Überlaufsatz seine zugewiesene Seite wieder verlassen muss, wird zunächst versucht, ihn wieder in seiner Hauseite unterzubringen. Falls nicht genügend Speicherplatz frei ist, erfolgt der neue Überlauf wieder von der Hauseite aus. Jeder Satz ist mit maximal zwei Seitenzugriffen aufzufinden. Da diese Indirektion jedoch das System verlangsamt (insbesondere wenn beide Seiten von der Platte zu laden sind) sollten möglichst wenige Datensätze aus ihren Hauseiten verdrängt werden. Hierfür kann man bei Systemen wie beispielsweise ORACLE beim Anlegen einer Tabelle definieren, dass ein gewisser Prozentsatz des Speichers einer Seite nicht für neue Datensätze genutzt werden darf, sondern nur für Änderungen vorbehalten ist. Dadurch können Datensätze ohne Verschieben wachsen. Natürlich muss man für eine sinnvolle Angabe dieses Freiplatzes das Anwendungsverhalten der Tabellen kennen.

11.4.2 Abbildung von Sätzen

Ein Datensatz muss meist vollständig in eine Seite passen. Im Allgemeinen sind jedoch mehrere Sätze in einer Seite untergebracht.

Jeder Satztyp (jede Tabelle) besitzt eine Formatbeschreibung, die im DB-Katalog (Data Dictionary) verwaltet wird. Sie setzt sich aus den Beschreibungen der Felder, aus denen der Satz aufgebaut ist, zusammen. Eine Feldbeschreibung besteht aus einer Liste seiner Eigenschaften wie

- ▶ Name
- ▶ Typ (alpha-nummerisch, nummerisch, binär, ...)

- Charakteristik (fest, variabel)
 - Länge (evtl. Maximum)
 - NULL zulässig

Die Formatbeschreibung steuert alle Operationen auf den Sätzen des entsprechenden Satztyps. Aus Gründen der Verarbeitungseffizienz und -flexibilität sollte die Speicherungsstruktur von Sätzen folgende wichtige Eigenschaften erfüllen:

- Jeder Satz hat eine eindeutige interne Kennung um beispielsweise bei Reorganisations- oder Recovery-Maßnahmen eine eindeutige Zuordnung zu erzielen.
 - Er sollte möglichst platzsparend gespeichert werden.
 - Eine Erweiterung des Satztyps muss im laufenden Betrieb möglich sein.
 - Die satzinterne Adresse des n-ten Felds sollte sich möglichst einfach berechnen lassen.

Die Diskussion möglicher Speicherungsformen wird an dem Satztyp PERS vorgestellt, der wie folgt aufgebaut ist.

PNR	NUMBER(5)	PRIMARY KEY
Name	VARCHAR(50)	NOT NULL
Beruf	VARCHAR(50)	
Gehalt	NUMBER(10,2)	NOT NULL
Ort	CHAR(2)	NOT NULL
AName	VARCHAR(50)	

Die einfachste Form der Abspeicherung als bloße Konkatenation von Feldwerten fester Länge führt auf Sätze fester Länge. Variabel lange Felder und undefinierte Feldwerte müssen in der im Data Dictionary angegebenen Maximallänge gespeichert werden. Die Verwaltung dieser Speicherungsform ist sehr einfach, ein großer Teil des Speicherplatzes wird jedoch unnötig belegt. Deshalb ist die Speicherung von Sätzen variabler Länge vorzuziehen.

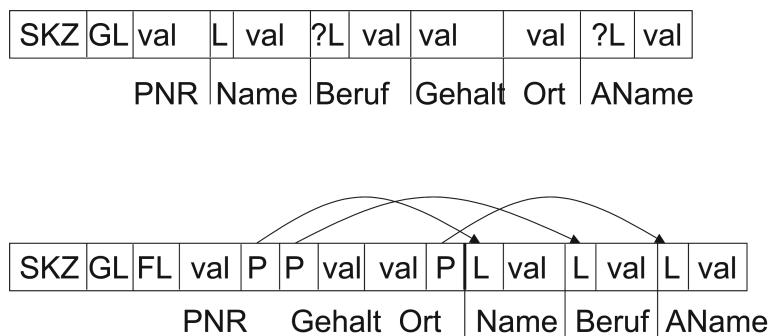


Abbildung 11.7: Speicherungsform mit Längenfeldern und Zeigern

Da Längenangaben für variable Felder nicht dem Data Dictionary entnommen werden können, sind sie jedem Satz als Strukturinformation (Längenfelder L) mitzugeben (Abbildung 11.7 oben, GL ist Gesamtlänge des Satzes). Damit NULL-Felder keinen Wert benötigen, sollte für diese auch kein Speicher belegt werden. Deshalb kann es zusätzliche Flags geben (hier ?), die anzeigen, ob der Datensatz bei dieser Spalte einen Wert hat oder undefiniert ist. Ist er undefiniert folgt direkt nach dem Flag der nächste Attributwert. Bei Variablen Feldern kann man diese Information mit

der Längenangabe definieren, wobei dann bei einzelnen Systemen keine Unterscheidung zwischen NULL oder einer Zeichenkette der Länge 0 mehr möglich ist. Bei Feldern fester Länge ist hierfür ein Zusatzbyte notwendig.

Offensichtlich wird hier der Grundsatz der Speicherökonomie beherzigt. Auch dynamische Erweiterbarkeit lässt sich leicht bewerkstelligen. Wird eine neue Spalte für die Tabelle hinzugenommen, muss dies nur im Katalog vermerkt werden. Beim Zugriff auf die Tabelle wird über die Gesamtlänge GL und die Einzellängen L erkannt, ob ein Wert für die neue Spalte vorhanden ist. Ist die nicht der Fall, ist der Wert für die Spalte undefiniert.

Wegen der eingestreuten Längenfelder ist jedoch keine direkte Berechnung der satzinternen Adressen aus den Katalogdaten möglich. Deshalb wird zu diesem Zweck die Optimierung nach Abbildung 11.7 unten vorgeschlagen. Dabei werden die variabel langen Felder durch Zeiger ans Ende des festen Strukturteils gelegt. Über Kataloginformationen kann nun die satzinterne Adresse für ein Feld fester Länge direkt und für eines variabler Länge bis auf eine Indirektion ermittelt werden. Die dynamische Erweiterbarkeit bleibt möglich, da fehlende Felder am Satzende prinzipiell erkannt werden können. Zur Vereinfachung lässt sich jedoch eine Längenangabe für den festen Strukturteil (FL) bereitstellen. Nach einer späteren Aktualisierung des Satzes mit dem neuen Feldwert sind Verschiebungen innerhalb der Speicherungsform durchzuführen.

Da viele Datenbankhersteller die Variante mit Längenangaben implementiert haben, sind Attributdefinitionen fester Länge und NOT NULL an den Anfang einer Tabellendefinition zu stellen. Dadurch ist die Position möglichst vieler Spalten direkt berechenbar.

Anhang A

Das Datenbank-Trainingssystem

A.1 Einführung in das Datenbanktrainingssystem DBTS

Die Swabia Beverage Holding AG besitzt mehrere Getränkemarkte im Bereich Ravensburg. Die einzelnen Getränkemarkte haben keine eigene Verwaltung, sondern werden durch die Swabia Beverage Holding AG von zentraler Stelle aus geführt. Zu diesem Zweck wurde eine Datenbank entworfen, die Mitarbeiter, Bestellungen und Lieferungen verwaltet. Die Swabia Beverage Holding AG legt großen Wert auf Kundennähe, deswegen sind auch Stammkunden erfasst, die über Liefertouren regelmäßig beliefert werden.

Ihre Aufgabe als Datenbankspezialist/in ist es nun, den Damen aus der Verwaltung der Swabia Beverage Holding AG bei ihren Abfragewünschen behilflich zu sein, da sie kein SQL beherrschen.

Vorschlag zur Vorgehensweise:

- ▶ Machen Sie sich vertraut mit der Datenbank, indem Sie sich das Datenbankmodell anzeigen lassen.
- ▶ Starten Sie das Training mit Aufgaben und gehen Sie der Reihe nach vor, manche Aufgaben basieren aufeinander.
- ▶ Die Datenbank läuft im Auto-Commit-Modus, deshalb ist kein Commit notwendig bzw. Rollback möglich.
- ▶ Im freien Training werden Ihnen keine Aufgaben vorgegeben. Hier können Sie eigene Statements ohne Hilfestellung ausführen.
- ▶ Es besteht eine Restoremöglichkeit, d. h. Sie können jederzeit die Originaldaten wiederherstellen.

A.2 Einrichten des DBTS

1. Installation

Kopieren Sie den Ordner xampplite auf Ihrem USB-Stick (bzw. Festplatte) direkt in das root-Verzeichnis, da es ansonsten zu Problemen beim Start des Apache Servers kommt. Falls Sie bereits im Besitz einer XAMPP-Version sind, reicht das Einfügen des Ordners 'datenbank' in das htdocs-Verzeichnis.

2. Autostart

Für das DBTS wird eine Autostart-Funktion angeboten. Wenn Sie DBTS auf einem USB-Stick betreiben, bietet Windows beim Einstecken des USB-Sticks den Start von DBTS an. Dazu muss sich im Rootverzeichnis die autorun.inf befinden. Falls kein Autostart gewünscht ist, kann die autorun.inf gelöscht werden.

3. Start und Beenden des XAMPP-Diensts

Wenn Sie keinen Autostart verwenden, können Sie XAMPP auch manuell starten. Verwenden Sie dazu die im Ordner xampplite vorhandene Datei `xampp_start.exe`. Durch einen Doppelklick startet der Apache- und MySQL-Server. Um zu überprüfen, ob der Server ordnungsgemäß zur Verfügung steht, geben Sie in einem Browserfenster `http://localhost` ein. Die Startseite von XAMPP sollte erscheinen. Ist dies nicht der Fall, entnehmen Sie bitte weitere Informationen aus dem Kapitel 'Umgang mit Fehlern'.

4. Beenden

Zum Beenden von XAMPP führen Sie die Datei `xampp_stop.exe` aus.

Empfehlung: Beenden Sie XAMPP immer, wenn Sie nicht mehr mit DBTS arbeiten und vor allem, wenn Sie Ihren Stick vom Rechner entfernen möchten.

A.3 Die Übungsoberfläche im Überblick

Die Startseite von DBTS erreichen Sie entweder automatisch durch den eingerichteten Autostart oder über den Aufruf `http://localhost/datenbank` in einem Browser. Grundsätzlich sind alle Oberflächen gleich strukturiert. Links befindet sich die Navigation, rechts daneben der Anzeigebereich. Auf der Startseite finden Sie folgende Navigationspunkte:

- ▶ **Startseite:** Zeigt die Startseite von DBTS an.
- ▶ **Datenbankmodell anzeigen:** Öffnet das Datenbankmodell in einem neuen Fenster.
- ▶ **Datenbank zurücksetzen:** Die Datenbank kann urgelaufen werden.
- ▶ **Training mit Aufgaben:** Führt zu den Übungsaufgaben des DBTS.
- ▶ **Freies Training:** Bietet die Möglichkeit, ohne konkrete Aufgabe auf der Datenbank zu arbeiten.

Wenn Sie sich entschieden haben, mit den zur Verfügung gestellten Aufgaben zu arbeiten, wählen Sie den Menüpunkt 'Training mit Aufgaben'. In der Navigation werden die Übungen, in entsprechende Gruppen eingeteilt, angezeigt. Weitere Informationen finden Sie dazu in Abschnitt 4. Der Aufbau des Anzeigebereichs ist in Abbildung A.1 dargestellt.

1. **Übungsfrage:** In diesem Bereich steht die Aufgabe, für die Sie ein passendes SQL-Statement als Lösung erarbeiten sollen.
2. **Ausgabefeld:** Es enthält alle Systemausgaben für den Benutzer. Dies kann entweder eine Tabelle, eine Fehlermeldung der Datenbank, ein Syntaxbeispiel oder die erwartete Lösung sein.
3. **Lösungsfeld:** Es dient zur Anzeige geforderten Lösung. In manchen Fällen gibt es auch nur eine textuelle Lösung!
4. **SQL Eingabefeld:** In diesem Textfeld werden SQL-Statements eingegeben.

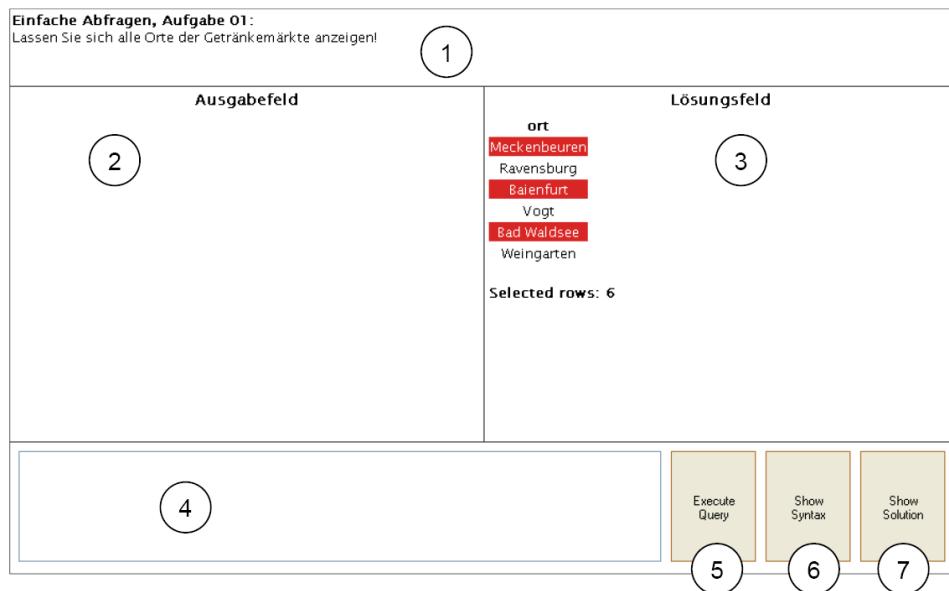


Abbildung A.1: Aufbau des Anzeigebereichs

5. **Execute Query Button:** Mit einem Klick wird der Inhalt des SQL Eingabefelds an die Datenbank gesendet und die entsprechende Rückgabe im Ausgabefeld dargestellt. (Shortcut: Alt + Shift + E)
6. **Show Syntax Button:** Mit einem Klick wird ein Syntaxbeispiel für das gesuchte SQL-Statement im Ausgabefeld angezeigt. (Shortcut: Alt + Shift + Y)
7. **Show Solution Button:** Mit einem Klick wird eine Musterlösung im Ausgabefeld angezeigt. (Shortcut: Alt + Shift + O)

A.4 Erläuterungen der SQL Abfragen

Insgesamt stellt das System 58 Übungen zur Verfügung. Diese wurden in sieben Gruppen unterteilt:

- ▶ **Einfache Abfrage:** Einfache Select-Abfragen, die als Einführung genutzt werden sollen, um mit dem Umgang am System vertraut zu werden.
- ▶ **Gruppenfunktionen:** Beinhaltet Select-Abfragen in Kombination mit Gruppenfunktionen
- ▶ **Join-Abfragen:** Umfasst Select-Abfragen, bei denen Tabellen miteinander verknüpft werden müssen.
- ▶ **Expertenaufgaben:** Schwierigere Aufgaben für fortgeschrittene Anwender.
- ▶ **Views:** Übungen rund um das Thema Views.
- ▶ **Datenmanipulation:** Bei diesen Aufgaben werden Datensätze modifiziert.
- ▶ **Tabelle anlegen:** Enthält Aufgaben, die sich mit dem Anlegen, Befüllen und Löschen einer Tabelle befassen.

A.5 Das Datenbankmodell

Im Navigationsmenü haben Sie die Möglichkeit, sich das Datenbankmodell anzeigen zu lassen. Sie erhalten einen Überblick über die vorhandenen Tabellen, ihre Attribute und Beziehungen (vgl. Abbildung A.2).

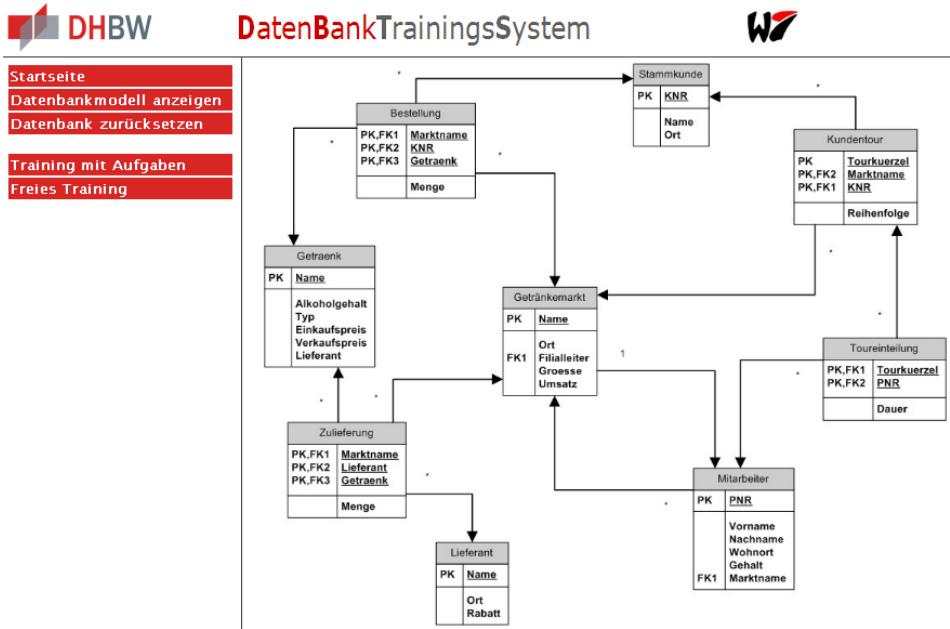


Abbildung A.2: Aufbau des Anzeigebereichs

A.6 Die Wiederherstellung der Datenbank

Möchten Sie aus verschiedenen Gründen eine Urladung vornehmen, so bietet Ihnen das System eine Wiederherstellung der Datenbank an. Wählen Sie dazu in der Navigation den Menüpunkt 'Datenbank zurücksetzen' aus. Mit einem Klick auf den Button 'Datenbank zurücksetzen' beginnt der Recoveryprozess. Dieser kann, vom Speichermedium abhängig, bis zu zwei Minuten in Anspruch nehmen. Warten Sie auf jeden Fall, bis unterhalb des Buttons eine Bestätigung der Aktion oder eine Fehlermeldung erscheint. Im Falle eines Fehlers besteht die Möglichkeit, die Urladung mit Hilfe von phpMyAdmin einzuspielen. Weitere Informationen zum Recovery mit phpMyAdmin finden Sie in Abschnitt 7 'Umgang mit Fehlern'.

Anhang B

Darstellungskonventionen

INSERT Schlüsselworte werden in Großbuchstaben dargestellt

Ausdruck Gemischte Groß/Kleinschreibung beschreibt den variablen Teil einer Anweisung

< > Platzhalter für erweiterte Syntaxbeschreibungen

[] Optionale Komponente

{ } Gruppe von Symbolen, Klammern sind nicht einzugeben

() Bestandteil der Syntax, sind anzugeben

| Alternative Termini werden durch dieses Symbol getrennt

... Terminus kann beliebig wiederholt werden