Modul

Datenbanken

Studiengänge Wirtschaftsinformatik/Praktische Informatik/Technische Informatik

Duale Hochschule Gera-Eisenach
- Campus Gera -

Autor

Stefan Dorendorf

Version 1.2 vom 10. September 2020

Hinweis

Dieses Skript enthält eine kurze Zusammenfassung der wesentlichsten Informationen zu den in der Lehrveranstaltung behandelten Themen. Es kann weder die Teilnahme an der Lehrveranstaltung noch das Studium einschlägiger Fachliteratur ersetzen.

Inhaltsverzeichnis

1 Einführung

1.1 Historie

Bis vor einigen Jahren wurde überwiegend der Begriff EDV (Elektronische Datenverarbeitung) für die kommerzielle Nutzung von Rechentechnik verwendet. Dieser Begriff zeigt deutlich die damalige Vorgehensweise bei der kommerziellen Rechneranwendung. Die Programme waren auf die Verarbeitung von relativ kleinen Datenmengen ausgelegt. Dabei war schon bei der Erfassung der Daten festgelegt, welche Informationen daraus gewonnen werden sollten.

Diese verarbeitungsorientierte Arbeitsweise hatte vor allem den Nachteil, dass die Daten und die Programme aufeinander fixiert waren.

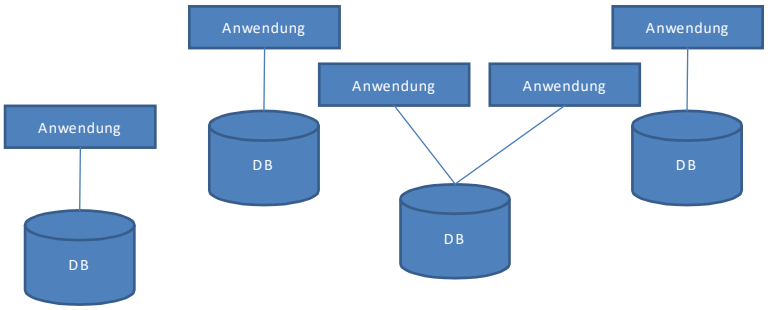


Abbildung 1: Insellösungen

Daraus ergaben sich entscheidende *Nachteile*:

1. *Redundanzen* (Datenbestandteile mehrfach abgespeichert): Diese können Inkonsistenzen zur Folge haben, das heißt, Änderungen in einem Datenbestand bleiben in allen anderen Beständen unberücksichtigt oder bedürften eines komplizierten Änderungsdienstes.

   → häufige Widersprüche zwischen den Datenbeständen möglich

2. *Mehrfachnutzung* von Datenverwaltungsfunktionen nicht möglich

   → oft wurde das Rad nicht nur zweimal erfunden

3. Eine *durchgängig rechnergestützte Verarbeitung* war oft nicht möglich.

4. Verstoß gegen die *Datenunabhängigkeit* durch die enge Kopplung von Programmen und Datenstruktur

   - Eine Datenmenge konnte nur verarbeitet werden, wenn deren logische und physische Struktur bekannt war.
   - Bei Nutzung der Daten von mehreren Programmen durfte die Datenstruktur nicht verändert werden, bzw. eine Änderung der Struktur des Datenbestands zog zwangsläufig eine Änderung in allen betroffenen Programmen nach sich.

Vorteil: kürzere Projektentwicklungszeiten, da nur spezielles Einzelproblem gelöst werden musste

Auch heute sind solche Insellösungen noch oft zu finden. Zwar werden durchaus häufig professionelle Datenverwaltungssysteme verwendet, trotzdem arbeiten die Anwendungen oft auf „ihren eigenen“ Datenbeständen. Die enge Verbindung zwischen Anwenderprogrammen und der Struktur des jeweiligen Datenpools ist oft noch vorhanden. Um die damit verbundenen Problem zu beseitigen (oder wenigstens abzumildern), wird ein Übergang von der *verarbeitungsorientierten Sicht* zur *informationsorientierten Sicht* angestrebt. Das bedeutet, dass für eine Gruppe von Benutzern ein einheitlicher Datenpool geschaffen wird, aus dem jeder die Informationen erhält, die er benötigt.

Merkmale des Wandels von der Verarbeitungsorientierung zur Informationsorientierung:

- Im Mittelpunkt der Informationsverarbeitung stehen nicht die Funktionen, sondern die Daten der Organisation.
- Im Gegensatz zu den Funktionen ändert sich das Datenmodell einer Organisation im Laufe der Zeit nicht wesentlich.
- Der Datenbestand ist als eine von seiner Verwendung unabhängige Ressource anzusehen.

Der grundlegende Unterschied zwischen der herkömmlichen Datenverarbeitung und dem Einsatz von Datenbanken liegt in der Sicht auf die Ressourcen eines Systems. Die informationsorientierte Datenverarbeitung sieht die Programme und Daten gleichberechtigt. Die herkömmliche Verarbeitung sieht als wichtigsten Punkt die Verarbeitung der Daten (die Programmierung) an sich.

Durch den Einsatz von Datenbanken, wird die Bedeutung der Daten als eigenständige Ressource hervorgehoben.

1.2 Grundlagen

1.2.1 Begriffe

Datenbank:

Eine Datenbank ist eine Menge von Daten, die weitestgehend redundanzfrei unter Steuerung eines Datenbank-Management-Systems (DBMS) abgespeichert, verwaltet und manipuliert wird. Eine Datenbank ist ein Modell der Informationen und Informationsstrukturen eines bestimmten Ausschnitts der realen Umwelt.

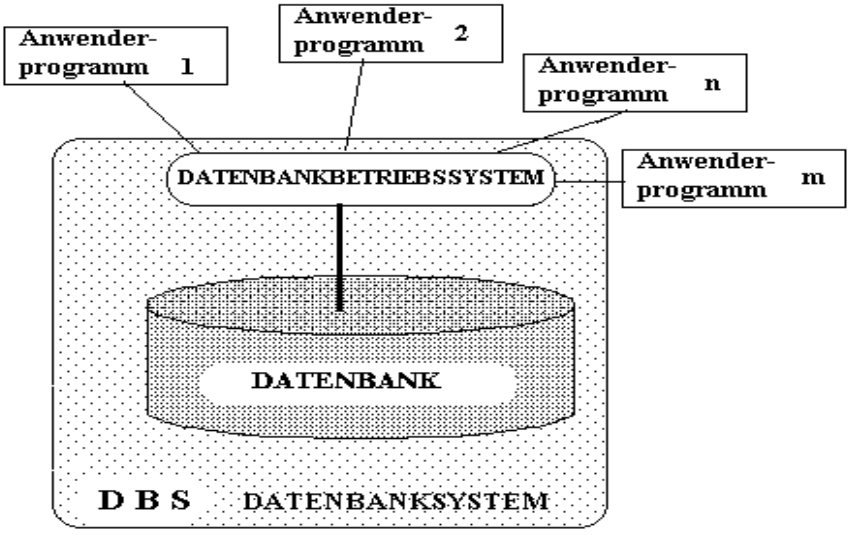


Abbildung 2: Datenbanksystem

Die Datenbank und ihr Datenbank-Management-System werden als Datenbanksystem bezeichnet.

Modell:

- Ein Modell ist ein Abbild eines Ausschnitts der objektiven Realität (Umwelt).
- Es ist das Ergebnis eines Prozesses. → des Modellbildungsprozesses
- Der Modellbildungsprozess ist zielgerichtet.

Die Vorteile der Nutzung von Datenbanken sind:

- Redundanzminderung
- zentrale Verwaltung und damit eine erhöhte Datensicherheit
- Trennung von Anwendungsprogrammen und Datenhandhabung
- Datenintegrität und Konsistenz
- Erreichen der Datenunabhängigkeit → Datenunabhängigkeit bedeutet die Trennung der Daten von den Programmen

physische Datenunabhängigkeit:

Ein Anwendungsprogramm ist von Änderungen in der physischen Abspeicherung der Daten unabhängig, das heißt, die Speicherstrukturen sind uninteressant.

logische Datenunabhängigkeit:

Die logische Sicht eines Programms und damit das Anwendungsprogramm selbst, bleibt unverändert in der Gesamtsicht der Daten.

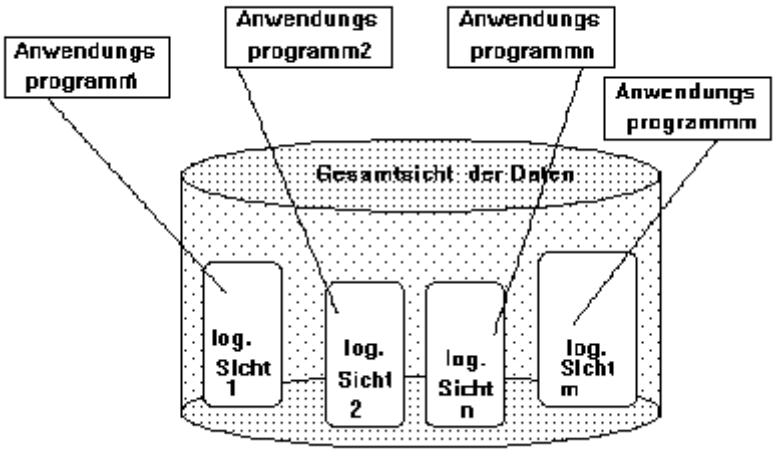


Abbildung 3: Sichten

Beispiel:

- In der Datenbank eines Unternehmens werden Daten für den Einkauf, die Lagerung und den Vertrieb von Artikeln gespeichert.
- Wenn sich jetzt z.B. die Sicht der Einkaufsabteilung ändert, weil zukünftig Daten über die Liefertreue von Lieferanten mit gespeichert werden sollen, so darf diese Erweiterung zunächst keine Auswirkungen auf die Anwendungen für Lagerhaltung und Vertrieb haben, wenn die Forderung nach logischer Datenunabhängigkeit eingehalten werden soll.

Als **Datenbank-Management-System** werden die Programme bezeichnet, die einzig und allein berechtigt sind, Operationen über den in der Datenbank gespeicherten Daten durchzuführen. Alle Anwendungsprogramme müssen über das DBMS auf die Daten zugreifen.

Aufgaben von DBMS

- Datendefinition
  - Definition der eigentlichen Daten
  - Definition der Zugriffshilfen
  - Definition der Benutzersichten
- Datenmanipulation
  - Retrieval - Operationen (wiederauffinden)
  - Update - Operationen (aktualisieren -> einfügen, löschen, ersetzen)
- Datenverwaltung
  - Laden und Entladen der Datenbank
  - Modifizieren der Speicherstrukturen
  - Reorganisation

- Datensicherheit
    - Zugriffskontrolle
    - Sicherung der Integrität
    - Synchronisation von Mehrfachzugriffen

Datenmodell:

Ein Datenmodell ist eine Modellierungsvorschrift die aus bestimmten Strukturtypen, festen Integrationsvorschriften für die Zuordnung der Strukturtypen zu Objekt- und Beziehungstypen und bestimmten Operationen über diesen Strukturtypen besteht.

Schema:

Ein Schema ist eine formalisierte Darstellung des Modells eines Umweltausschnittes (grafisch oder normalsprachlich), hergeleitet aus den Vorschriften eines Datenmodells. Das Abbilden von Objekten eines höheren Schemas auf ein anderes, meist tieferes Schema, wird als Schema-Mapping bezeichnet.

1.2.2 Architekturen von DBMS

In diesem Abschnitt wird die Architektur von Datenbank-Management-Systemen zunächst im Überblick behandelt. Detailliertere Ausführungen zu ausgewählten Teilen finden sich noch in den folgenden Kapiteln bzw. in [HR01].

Für die Trennung von Daten und Anwendungsprogrammen wurde 1975 vom Standards Planning and Requirements Committee (SPARC) des American National Standards Institute (ANSI) 3-Ebenen-Konzept entwickelt.

In der **externen Ebene** wird mit Hilfe von geeigneten (logischen / externen) Datenstrukturen festgelegt, wie den Anwendungsprogrammen der Benutzer die Daten der Datenbank präsentiert werden sollen, unabhängig von ihrer internen Struktur. Ein externes Schema beschreibt die Sichtweise einer Anwendung auf die Daten. Es kann durchaus zu jedem Anwendungsprogramm ein eigenes externes Schema geben, das die Daten in der Weise darstellt, wie es das jeweilige Programm erwartet.

In der **konzeptuellen Ebene** stellt das konzeptuelle Schema (Realitätsmodell) das Informationsangebot (logische Gesamtstruktur der Daten, ihre Eigenschaften und ihre Beziehungen untereinander) auf abstrakter Ebene dar. Es ist neutral gegenüber einzelnen Anwendungen und deren Sicht auf die Daten. Hier gilt derzeit das relationale Modell quasi als Standard.

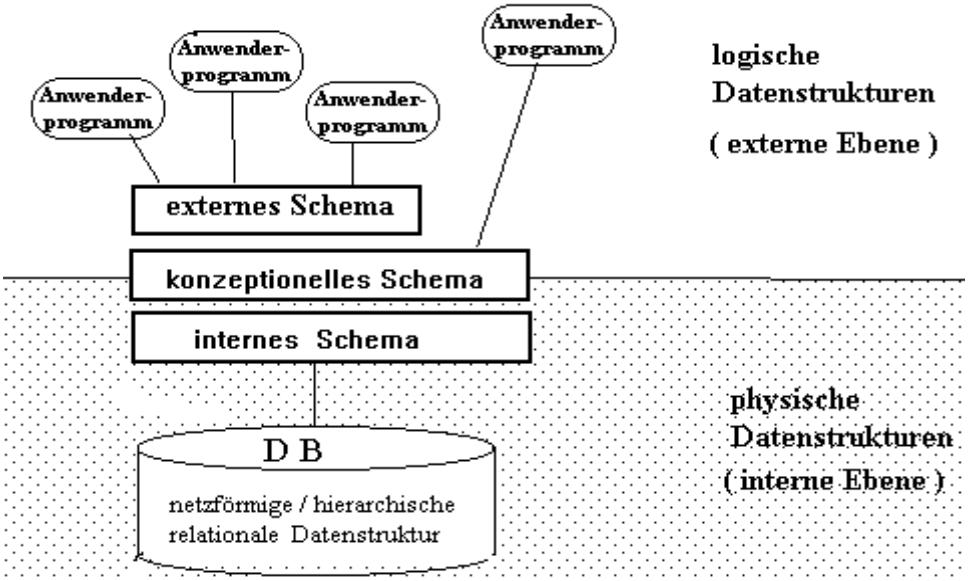


Abbildung 4: Architekturvorschlag nach ANSI-SPARC

In der **internen Ebene** wird mit geeigneten physischen Datenstrukturen festgelegt, wie die Daten auf externen Speichermedien (z.B. Festplatten) zu speichern sind. Dabei wird sowohl auf das Realitätsmodell der konzeptuellen Ebene, als auch auf die hardwarespezifischen Eigenheiten Bezug genommen.

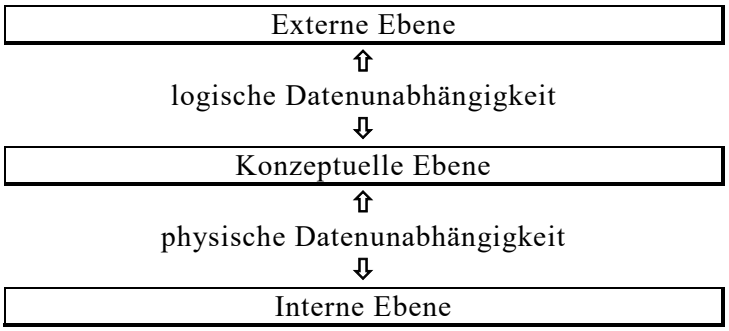


Abbildung 5: Erreichen von Datenunabhängigkeit durch 3-Ebenen-Architektur

Datenbank-Management-Systeme verfügen über Komponenten, die die Zugriffe auf die gespeicherten Daten realisieren. Um Leistungen, wie z.B. Datenunabhängigkeit oder die Sicherung der Konsistenz von Daten, die von DBMS ebenfalls gefordert werden, erbringen zu können, müssen die Systeme Beschreibungsinformationen zu den von ihnen verwalteten Daten (Metadaten) führen und Mechanismen zur Verwaltung von Transaktionen realisieren. Die wichtigsten Komponenten von DBMS zeigt *Abbildung 6*.

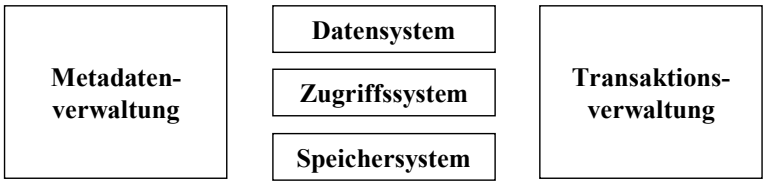


Abbildung 6: Grober Überblick über die Komponenten von DBMS

Große Softwaresysteme (zu denen typischerweise auch DBMS zählen) werden meist zur Reduzierung der Komplexität in mehreren Schichten/Ebenen (Layer) implementiert. Durch Modularisierung werden spätere Anpassungen und Erweiterungen deutlich erleichtert. In Systemen mit Schichtenarchitekturen gelten i.d.R. folgende Prinzipien:

- Die höheren Schichten nutzen die Leistungen der tieferen Schichten.
- Änderungen an höheren Schichten wirken sich nicht auf tiefere Schichten aus.
- Die höheren Schichten können abgetrennt werden, trotzdem bleiben die tieferen Schichten funktionstüchtig.
- Die tieferen Schichten können ohne die höheren Schichten getestet werden.
- Programme einer bestimmten Schicht verwenden die darunter liegende Schicht als „virtuelle Maschine“.

Jede Schicht besitzt dabei Schnittstellen zur jeweils unter- bzw. übergeordneten Schicht und wird hier aus Datenobjekten und Operatoren gebildet. *Abbildung 7* zeigt zunächst grob eine mögliche Schichtenarchitektur, mit der Zugriffe auf die Daten einer Datenbank realisiert werden können.

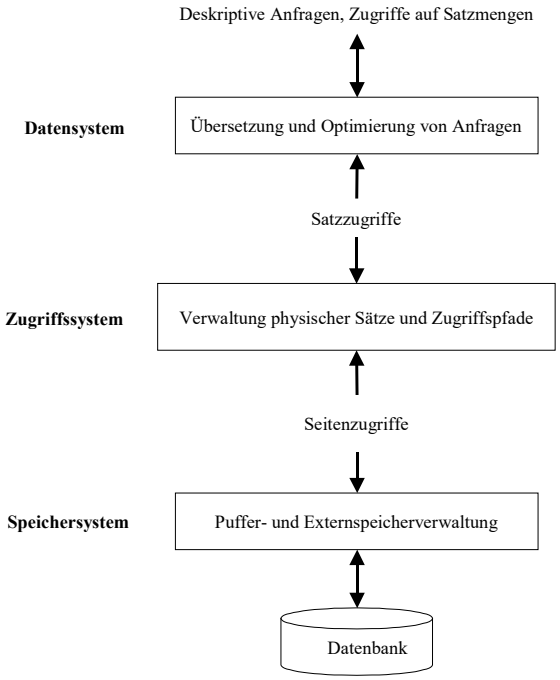


Abbildung 7: Einfaches Schichtenmodell (vgl. [HR01])

Durch weitere Verfeinerung des einfachen Schichtenmodells ergibt sich das in *Abbildung 8* dargestellte gängige Modell zur Darstellung des allgemeinen Aufbaus von DBMS, dessen Komponenten kurz beschrieben werden sollen. Das Modell beschreibt auch die *Transformation von Daten*, ausgehend vom Speicherformat auf externen Speichermedien, bis hin zur mengenorientierten Darstellung, wie sie von Anwendungsprogrammen üblicherweise verwendet wird.

- Die oberste Schicht bilden die **Transaktionsprogramme** (z.B. Benutzerprogramme), die auf die in der jeweiligen Datenbank gespeicherten Daten zugreifen. Sie nutzen dazu üblicherweise die von DBMS zur Verfügung gestellten mengenorientierten Schnittstellen (Tabellen, Sichten, Tupel) unter Nutzung deskriptiver Zugriffsmöglichkeiten (SQL, QBE, ...).

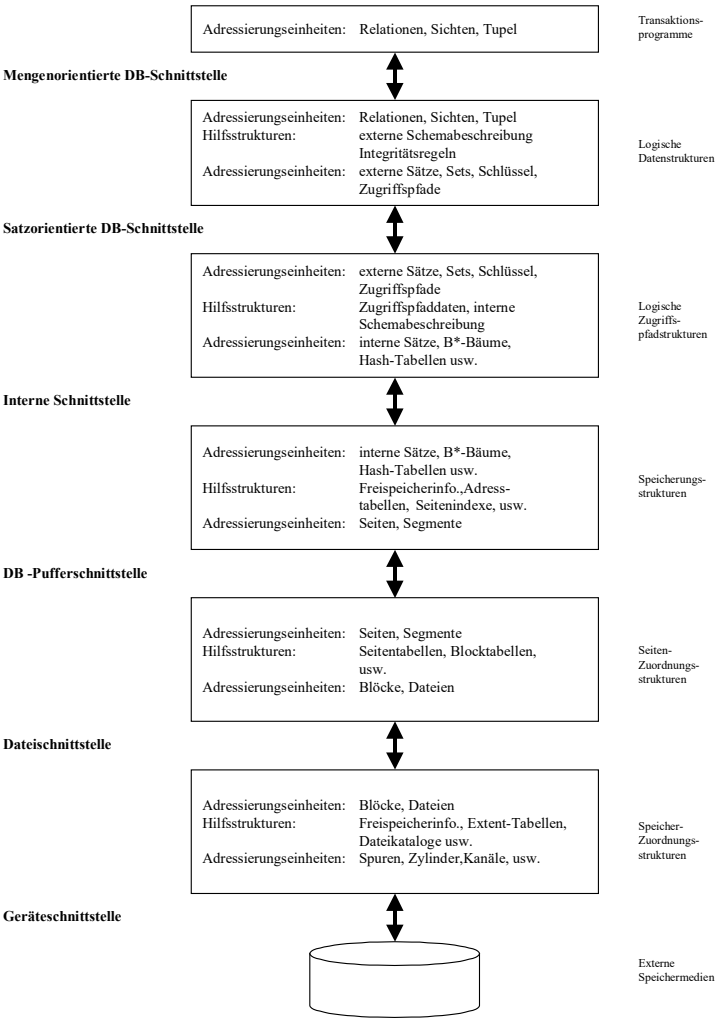


Abbildung 8: Schichtenmodell eines DBMS (nach [HR01])

- Die nächste Schicht realisiert die **logischen Datenstrukturen**, auf die die Anwendungsprogramme (im weitesten Sinn) zugreifen. Sie stellt nach oben hin eine mengenorientierte Schnittstelle (Treffermengen) sowie entsprechende Zugriffsmechanismen (deskriptiv, z.B. über SQL) zur Verfügung. Weiterhin werden von dieser Schicht externe Schemabeschreibungen verwaltet, Zugriffskontrollen realisiert und die Einhaltung von Integritätsregeln sichergestellt. Zur Realisierung der Schicht wird die satzorientierte Datenbankschnittstelle (externe Sätze – z.B. ohne Längen-Bytes etc. –, Sets, Schlüssel, Zugriffspfade) genutzt.
- Die satzorientierte Datenbankschnittstelle (Datensätze, Indexeinträge) wird von der Schicht der **logischen Zugriffspfadstrukturen** realisiert. Die Schicht

verwaltet interne Schemabeschreibungsinformationen (z.B. über die konkrete Realisierung der jeweiligen Zugriffspfade) und realisiert u.a. einfache Operatoren für (tw. auch navigierende) Datenzugriffe (z.B. `FIND NEXT <satz>`, `STORE <satz>`) und die Verwaltung von Transaktionen. Die Schicht greift auf die interne Satzschnittstelle (interne Sätze, B*- Bäume, Hash-Tabellen usw.) zu.

- Die nächste Schicht realisiert die vom konkreten DBMS verwendeten **Speicherungsstrukturen** (Heap, B*-Bäume, Cluster, IOT, ...) und stellt nach oben hin die interne Satzschnittstelle zur Verfügung. Hier werden Operationen wie das Finden, Einfügen oder Löschen von Sätzen/Einträgen in konkrete Daten- oder Indexstrukturen realisiert. Auch die Verwaltung von Sperren sowie Log- und Recovery-Mechanismen werden hier implementiert. Von der Schicht werden u.a. Freispeicherinformationen, Adresstabellen, Seitenindizes usw. verwaltet. Die Schicht greift auf die Schnittstelle zum Datenbankpuffer (z.B. Table Spaces, Segmente, Extents, Blöcke, Seiten) zu.
- Die Schicht zur Implementierung der **Seitenzuordnungsstrukturen** realisiert im Wesentlichen den Datenbankpuffer. Sie beinhaltet bspw. Operationen zum Reservieren, Freigeben, Auffinden von Seiten im Datenbankpuffer. Dazu werden Verwaltungsinformationen wie Seiten- bzw. Blocktabellen gehalten. Weiterhin wird in dieser Schicht die konkrete Seitenersetzungsstrategie des DBMS sowie Strategien zum Einbringen von Änderungen in die Blöcke auf den Datenträgern implementiert. Nach unten hin wird auf eine Dateischnittstelle (Dateien, Blöcke) zugegriffen.
- Die **Speicherzuordnungsstrukturen**, die in der nächsten Schicht implementiert sind, realisieren nach oben hin eine Dateischnittstelle. Die Schicht realisiert Operationen zum Lesen und Schreiben von Blöcken von bzw. auf die externen Speichermedien. Dazu werden Freispeicherinformationen, Dateikataloge, Extent-Tabellen usw. verwendet. In dieser Schicht erfolgt auch die Zuordnung von Blöcken innerhalb von Dateien auf die Adressierungseinheiten der Geräteschnittstelle (z.B. Kanäle, Zylinder, Spuren, Sektoren). Die Dateischnittstelle kann vom Betriebssystem zur Verfügung gestellt werden, aber auch eine nahezu direkte Verwaltung des Platzes auf den Speichermedien durch das DBMS ist möglich (z.B. durch die Verwendung von RAW-Devices).
- Die **externen Speichermedien** sind jeweils mit einer Geräteschnittstelle ausgerüstet, über die der Zugriff auf die Medien erfolgen kann. Dazu werden üblicherweise Kanalprogramme, Geräte-Driver etc. verwendet.

1.2.3 Datenzugriffe

Während das in *Abschnitt 1.2.2* beschriebene Modell im Wesentlichen den Aufbau von DBMS und Datentransformationen beschreibt, zeigt *Abbildung 9* die Abläufe bei der Verarbeitung von Anfragen, wie sie z.B. von Anwendungsprogrammen an das DBMS gestellt werden.

1. Zunächst stellt das Anwenderprogramm eine Anforderung (z.B. in Form einer SQL-Anweisung) an das DBMS.
2. Um die Ausführbarkeit der Anforderung zu prüfen und evtl. die Ausführung vorzubereiten, werden Datenbeschreibungsinformationen (Metadaten) benötigt. Anschließend erfolgt die Analyse der Anforderung. Ist diese syntaktisch korrekt formuliert und ausführbar, so wird eine Strategie zur kostengünstigen Ausführung (Ausführungsplan) festgelegt (Anfrageoptimierung).
3. In den nächsten Schritten werden die benötigten Datenblöcke angefordert.
4. Die Pufferverwaltung entscheidet, ob Blöcke von Datenträgern gelesen werden müssen und legt die Seitenrahmen fest, in denen die benötigten Blöcke im Datenbankpuffer zur Verfügung gestellt werden. Das Betriebssystem liest, wenn erforderlich, die benötigten Datenblöcke von den Datenträgern und stellt sie zur Verfügung.
5. Das DBMS bereitet die benötigten Daten auf und überträgt die aufbereiteten Daten in einen dem Anwendungsprogramm zugeordneten Speicherbereich (User Working Area – UWA).
6. Abschließend liefert das DBMS eine Meldung über die Ausführung der Anforderung und stellt die Daten bereit.

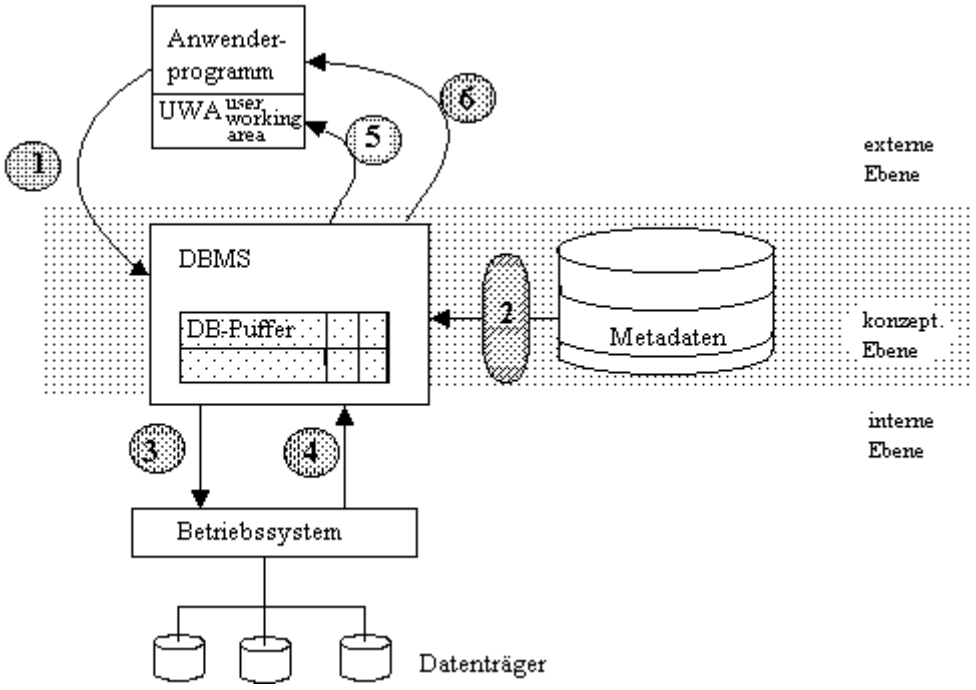


Abbildung 9: Schematische Darstellung der Abläufe bei Datenzugriffen (vgl. [TL91])

2 Datenmodelle

In diesem Kapitel werden verschiedene Datenmodelle betrachtet. Der Schwerpunkt liegt dabei klar auf dem relationalen Datenmodell, auf dem eine große Anzahl von Softwarelösungen basiert und das auch überwiegend als Basis für Neuentwicklungen von Anwendungen verwendet wird. Weiterhin werden vorrelationale Modelle, wie das hierarchische Datenmodell und das Netzwerkdatenmodell behandelt. Auch Erweiterungen von relationalen Systemen um Konzepte der Objektorientierung werden in die Betrachtungen mit einbezogen.

2.1 Hierarchisches Datenmodell

Das hierarchische Datenmodell ist das älteste der klassischen Datenmodelle. Das Produkt IMS (IBM Information Management System) ist ein DBMS, das auf dem hierarchischen Datenmodell basiert.

Grundelemente des Modells sind Datensätze (*records*), die Felder (*fields*) enthalten und die Daten der Objekte aufnehmen, über die Informationen in der jeweiligen Datenbank gespeichert werden sollen. Datensätze werden zu benannten Datensatzmengen zusammengefasst. Die Datensatzmengen sind hierarchisch in Eltern-Kind Beziehungen (*parent-child-relationships*) angeordnet. Dadurch ergeben sich Baumstrukturen. Die Datensätze werden in den Knoten der Bäume gespeichert. Die Beziehungen zwischen den Objekten werden durch die Kanten der Bäume dargestellt.

Zwischen den Datensätzen der Eltern- und den Kind-Datensatzmengen bestehen 1:N Beziehungen, d.h. einem Eltern-Datensatz können mehrere Kind-Datensätze zugeordnet sein. Diese Eltern-Kind Beziehung ist implizit gegeben und trägt somit keine explizite Bezeichnung.

Die Strukturen weisen folgende Eigenschaften auf:

- Es gibt eine ausgezeichnete Datensatzmenge als oberstes Element in der Hierarchie, d.h. sie ist nicht Kind-Datensatzmenge irgendwelcher Eltern-Datensatzmengen. Diese Datensätze stellen jeweils die Wurzel von Baumstrukturen dar.
- Jede Datensatzmenge, mit Ausnahme des obersten Elements, ist eine Kind-Datensatzmenge in genau einer Eltern-Kind-Beziehung. In einer Baumstruktur besitzt jeder Knoten, mit Ausnahme des Wurzelknotens, genau einen Vorgänger (Vaterknoten).
- Eine Datensatzmenge kann als Eltern-Datensatzmenge in mehreren (auch null) Eltern-Kind Beziehungen stehen. Ein Knoten einer Baumstruktur kann auf mehrere Nachfolger (Sohnknoten) verweisen. Knoten ohne Nachfolger werden als Blätter bezeichnet.

Abbildung 10 zeigt beispielhaft, wie ein Ausschnitt der Daten der Berufsakademie Gera auf das hierarchische Datenmodell abgebildet werden könnte.

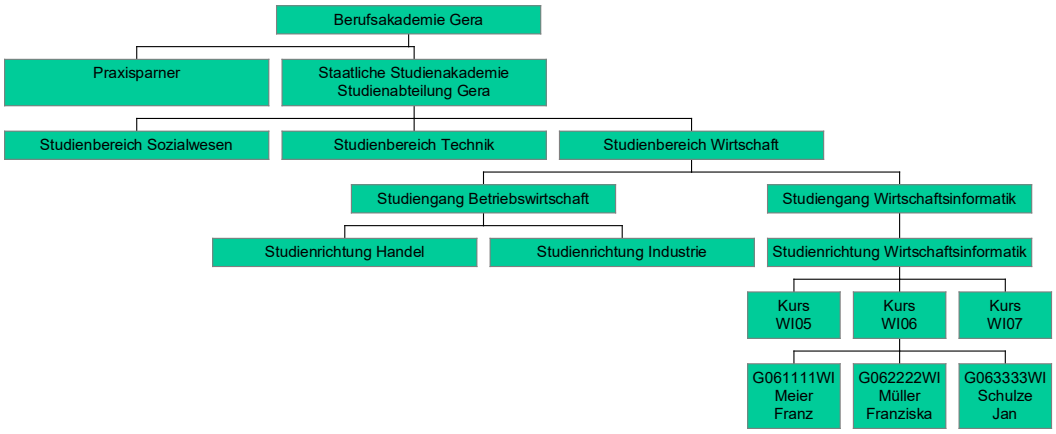


Abbildung 10: Beispiel Berufsakademie im hierarchischen Datenmodell

Möglichkeiten zur Datenspeicherung

Das hierarchische Datenmodell orientiert sich stark an den Konzepten traditioneller Dateisysteme. Jede Datensatzmenge wird üblicherweise in einer eigenen Datei gespeichert. Es ist aber auch möglich, ganze Baumstrukturen innerhalb einzelner Dateien abzulegen. Die normalerweise stets gleiche Größe der Sätze innerhalb einer Datensatzmenge erlaubt einen schnellen Zugriff auf einzelne Datensätze durch die Berechnung von Sprungadressen.

Die Beziehungen werden über Zeiger implementiert („hart verdrahtet“). Ein Datensatz aus der Eltern-Datensatzmenge zeigt dabei auf den ersten zugeordneten Datensatz aus der Kind-Datensatzmenge, deren Elemente wiederum untereinander verkettet sind.

Zur Beschleunigung des Zugriffs werden die Datensatzmengen oftmals indexiert. Dazu werden Indexe erstellt, die schnell durchsucht werden können und die direkt auf die Speicherorte von Datensätzen verweisen. Das hierarchische Datenbanksystem von IBM (das seit Ende der 1960er entwickelt wurde) ist tw. auch noch heute bei IBM-Kunden im Einsatz.

Grenzen des hierarchischen Modells

Die Verwendung von Zeigern zur Darstellung von Beziehungen hat den Vorteil, dass auch in sehr großen Datenmengen, die mit einem Eltern-Objekt in Beziehung stehenden Nachfolgeobjekte („Kinder“, „Enkel“, usw.) sehr schnell aufgefunden werden können. Wenn die Suchwege innerhalb eines Datenbestands vorab bekannt sind, so können Hierarchien erstellt werden, die diese Suchen unterstützen. Müssen neue Suchmöglichkeiten integriert werden, so müssen neue Strukturen (Bäume) in das Datenmodell aufgenommen werden. Der Preis für die auch innerhalb großer Datenmengen schellen Suchoperationen ist ein allerdings relativ starres Datenmodell.

Ein weiteres Problem ergibt sich daraus, dass es in Baumstrukturen nicht möglich ist, dass verschiedene Eltern-Objekte auf die selben Kind-Objekte verweisen. Damit sind im reinen hierarchischen Datenmodell n-m Beziehungen nicht zu modellieren. Sollen im Beispiel aus Abbildung 10 die Studierenden auch den Unternehmen zugeordnet werden, so müssen deren Datensätze unterhalb der Unternehmen erneut eingefügt werden, was zu Redundanzen führt.

2.2 Netzwerkmodell

Das Netzwerkmodell wurde von der *Data Base Task Group* (DBTG) des *Programming Language Committee* (später COBOL Committee) der *Conference on Data Systems Language* (CODASYL) vorgeschlagen. Diese Organisation war auch für die Definition der Programmiersprache COBOL verantwortlich. Es ist auch unter dem Namen "CODASYL Datenbankmodell" oder "DBTG Datenbankmodell" bekannt und stark von Cobol beeinflusst. Der DBTG-Bericht von 1971 enthielt Vorschläge für drei verschiedene Datenbanksprachen:

- eine Schema Data Description Language (Schema-Datenbeschreibungssprache)
- eine Subschema Data Description Language (Subschema-Datenbeschreibungssprache)
- eine Data Manipulation Language (Datenmanipulationssprache)

Das Netzwerkmodell beinhaltet Datensatz- und Mengentypen. Ein Datensatztyp hat einen Namen und umfasst Datensätze gleicher Struktur, die auch mehrwertige Attribute (sogar mehrwertige komplexe Attribute, *repeating collections*) enthalten können. Mengentypen dienen zur Darstellung von 1:N-Beziehungen zwischen einem Besitzer-(*owner*)Datensatztyp und einem Mitglieds-(*member*)Datensatztyp und werden im Gegensatz zur Eltern-Kind Beziehung im hierarchischen Datenmodell benannt. Die Elemente eines Mengentyps sind sortiert.

Das Netzwerkmodell fordert keine strenge Hierarchie. Es erlaubt die Verknüpfung zwischen beliebigen Datensatztypen durch Mengentypen. Damit können (zumindest indirekt) auch M:N-Beziehungen abgebildet werden. Dies erfordert Hilfsdatensatztypen (*dummy* oder *linking record type*), die zwei 1-N Beziehungen darstellen.

Oft existieren unterschiedliche Suchwege, um zu einem bestimmten Datensatz zu gelangen. Das Netzwerkmodell kann als Verallgemeinerung des hierarchischen Datenmodells aufgefasst werden, bei dem zumindest das Problem häufig auftretender Redundanzen vermindert wird.

Abbildung 11 zeigt am Beispiel eines Ausschnitts aus dem möglichen Datenmodell einer Berufsakademie die Abbildung der entsprechenden Informationsstrukturen auf das Netzwerkmodell.

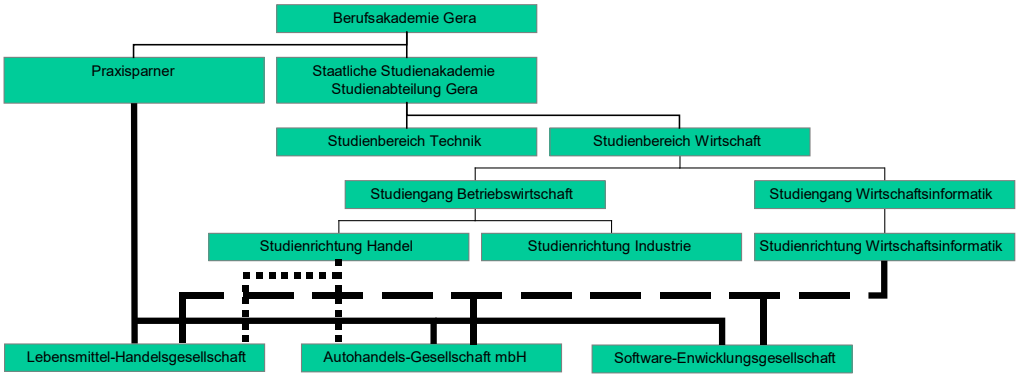


Abbildung 11: Beispiel Berufsakademie im hierarchischen Datenmodell

Hocheffiziente Netzwerkdatenbanksysteme existieren (existierten) auf mittleren und großen Mainframes, die für höchste Transaktionsraten ausgelegt waren bzw. sind, bis relationale Datenbanksysteme bezüglich der Performance einigermaßen gleichziehen konnten. Auch Abfragesprachen für Ad-hoc-Anfragen standen auf Netzwerksystemen zur Verfügung, beispielsweise QLP/1100 von Sperry Rand.

Bekannte Vertreter des Netzwerkdatenbankmodells sind UDS (Universal Datenbank System) von Siemens, DMS (Database Management System) von Sperry Univac. Mischformen zwischen relationalen Datenbanken und Netzwerkdatenbanken wurden entwickelt - z.B. von Sperry Univac (RDBMS Relational Database Managementsystem) und Siemens (UDS/SQL), mit der Absicht, die Vorteile beider Modelle zu verbinden.

Seit den 1990er Jahren wird das Netzwerkdatenbankmodell vom relationalen Datenbankmodell mehr und mehr verdrängt.

2.3 Das relationale Datenmodell

Das Datenmodell wurde von Edgar F. Codd 1970 erstmals vorgeschlagen und ist heute ein üblicher Standard zum Speichern von Daten. Obwohl das relationale Datenmodell und das Netzwerkdatenmodell nahezu zeitgleich vorgeschlagen wurden, dauerte die Etablierung relationaler Systeme wesentlich länger. Der Grund dafür stellt einen sicherlich wesentlichen Vorteil des Modells dar. Beziehungen zwischen Datenobjekten werden nicht mehr über Zeiger „fest verdrahtet“ sondern über deren Eigenschaftsausprägungen (Attributwerte) dargestellt. Dies sorgt für wesentlich flexiblere Auswertungsmöglichkeiten als beim Netzwerk- oder beim hierarchischen Modell, führt aber auch dazu, dass der Aufwand zur Ausführung der Auswertungen höher ist. Die Verbreitung relationaler Datenbanksysteme nahm daher erst mit der Verbreitung leitungsfähiger Hardware zu. Bekannt im Zusammenhang mit relationalen Datenbanken ist auch die Datenbanksprache SQL (Structured Query Language), zum Abfragen und Manipulieren von Daten sowie zur Datendefinition.

Für rein relationale Systeme ist die Beschreibung der logischen Datenstrukturen recht einfach. Daten werden in einfacher Tabellenform, ohne mengenwertige oder zusammengesetzte Attribute dargestellt. Weder die Reihenfolge der Attribute noch

die der Tupel einer *Tabelle* sind vorgeschrieben. Eine Tabelle kann daher auf eine ungeordnete Folge von Datensätzen abgebildet werden. Um Zugriffe auf einzelne Sätze bzw. kleinere Satzmengen zu beschleunigen, können häufig für Suchoperationen verwendete Attribute bzw. Attributkombinationen (*Suchschlüssel*) indexiert werden. Über den *Index* wird eine Zuordnung zwischen Suchschlüsselwerten und den Speicherorten der den jeweiligen Suchschlüsselwert enthaltenden Datensätze vorgenommen. Damit werden zusätzliche Zugriffspfade geschaffen. Tabellen stellen aus mathematischer Sicht Relationen dar.

mathematische Definition einer Relation :

Eine Relation zwischen zwei Mengen A und B ist eine Teilmenge des kartesischen Produkts von A und B.

$$R \subset A \times B$$

Das heißt, ein Objekt entspricht einer Zeile einer Tabelle (*n-Tupel*). Die Attribute von Objekten werden in den Tabellenspalten dargestellt. Ein Beispiel zeigt *Abbildung 12*.

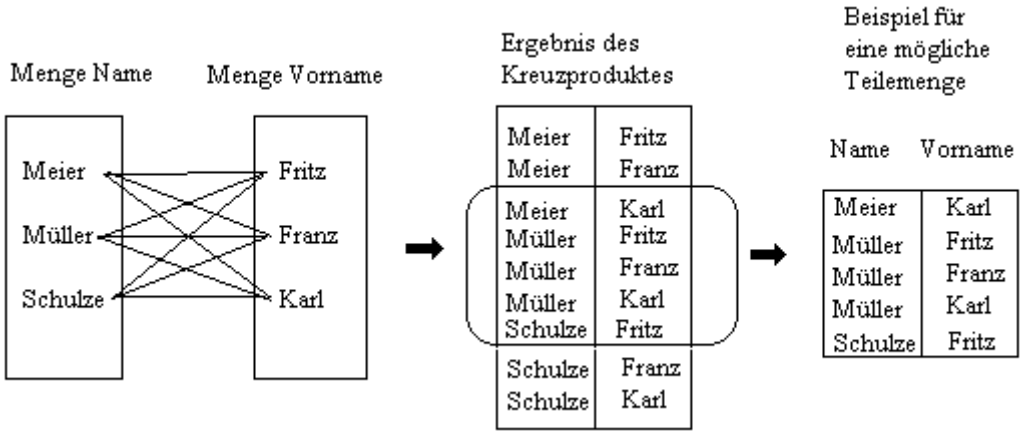


Abbildung 12: Bildung von Relationen

Im relationalen Datenmodell werden elementare Beziehungen zwischen Merkmalen (Attributen) durch die Bildung von Tabellen (Relationen) ausgedrückt. Eine bestimmte Wertekombination einer Tabelle beschreibt ein Datenobjekt (Entity). Beziehungen zwischen Entities werden durch korrespondierende Attributwerte in verschiedenen Tabellen (Schlüssel-Fremdschlüssel) – also noch auf der logischen Ebene – abgebildet. Änderungen der Beziehungen einzelner Entities untereinander werden durch Werteänderungen in den entsprechenden Attributen realisiert.

Im Zusammenhang mit dem relationalen Datenmodell sind die folgenden Festlegungen und Begriffe von Bedeutung.

- Ein Objekt hat bestimmte Merkmale (*Attribute*).
- Es existieren Beziehungen zwischen diesen Objekten.
- Diese Beziehungen haben wiederum bestimmte Merkmale.

Der Wertebereich eines Attributs wird als *Domäne* bezeichnet.

Eine *Entität* (*entity*) ist eine ganz bestimmte Kombination von Attributwerten für ein Objekt (z.B. Müller Karl).

Durch Klassenbildung (Verallgemeinerung) lassen sich mehrere Entities zu einem *Entitätstyp* (*entity type*) zusammenfassen (z.B. haben alle Objekte Person einen Namen und Vornamen).

Alle Entities, die mit einem Entity-Typ beschrieben werden können, sind eine *Entitätsmenge* (*entity set*) (Meier Karl, Müller Fritz, Müller Franz, Müller Karl und Schulze Fritz).

Die *Beziehungen* (*relationships*) zwischen Objekten (Entitäten) werden durch Verallgemeinerung zu *Beziehungstypen* (*relationship type*) zusammengefasst.

Alle Beziehungen, die durch einen Beziehungstyp beschrieben werden können, sind eine *Beziehungsmenge* (*relationship set*).

Die Begriffszuordnungen soll *Abbildung 13* verdeutlichen.

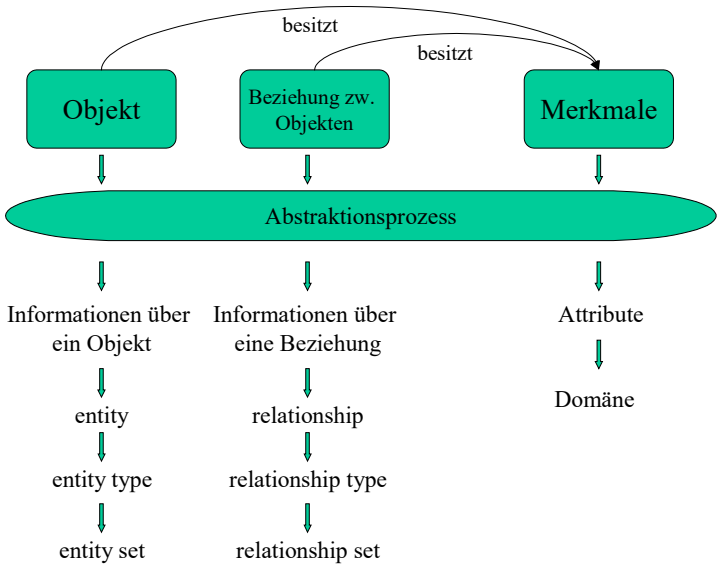


Abbildung 13: Begriffszuordnung für das relationale Datenmodell (vgl. [TL91])

Operationen des Relationalen Datenmodells

Die Operationen über relationalen Datenbanken basieren auf der Relationalen Algebra, die auf Mengenoperationen aufbaut. Wichtige Operationen sind:

- Selektion
- Projektion
- Join

Die *Selektion* wählt (selektiert) aus einer Relation n-Tupel (Datensätze) aus, für die eine bestimmte Selektionsbedingung erfüllt ist, das heißt, ein Prädikat (Selektionsprädikat) ist erfüllt.

Sei R eine Relation über $\{A_1, \ldots, A_k\}$ und B ein beliebiger Ausdruck. Die Selektion über R ist wie folgt definiert:

$$R[A_i \otimes B] := \{r \mid r \in R \wedge [A_i \otimes B] = true\} \quad mit \quad \otimes = \{=, <, >\leq, \geq, <>\} \quad und \quad i = \{1, \ldots, k\}$$

Handelt es sich im konkreten Fall bei B um einen konstanten Wert bzw. einen konstanten Ausdruck, so wird der Vergleichsausdruck als *monadischer Vergleichsterm* bezeichnet. Enthält B Attributbezeichner, so handelt es sich bei dem Vergleichsausdruck um einen *dyadischen Vergleichsterm*. Mehrere Selektionsbedingungen können mittels logischer Operatoren verknüpft werden. *Abbildung 14* soll zur Verdeutlichung dienen.

Attribut 1	Attribut 2	Attribut 3

Abbildung 14: Selektion

Die *Projektion* liefert Ergebnisrelationen, deren Typ auf einer echten Attributteilmenge der Ausgangsrelation basiert. Durch eine Projektion werden bestimmte Spalten einer Relation ausgewählt (*Abbildung 15*). Die Projektion über R ist wie folgt definiert:

$$R[\beta] := \{r_\beta \mid r \in R\} \quad \text{mit } \beta \subseteq \{A_1, \ldots, A_k\}$$

Attr.1	Attr.2	Attr. 3	Attr. 4	Attr. 5	Attr. 6	Attr. 7

Abbildung 15: Projektion

Die *Join*-Operation verbindet zwei Relationen zu einer neuen Relation in Abhängigkeit der Wertebelegung der Join-Attribute. Die Join-Attribute sind Attribute, deren Wertebereiche (Domänen) vergleichbar sind. Für zwei Relationen $R(A_1,\ldots,A_4)$ und $S(A_1,\ldots,A_{III})$ ist das Ergebnis einer Join-Operation in *Abbildung 16* dargestellt.

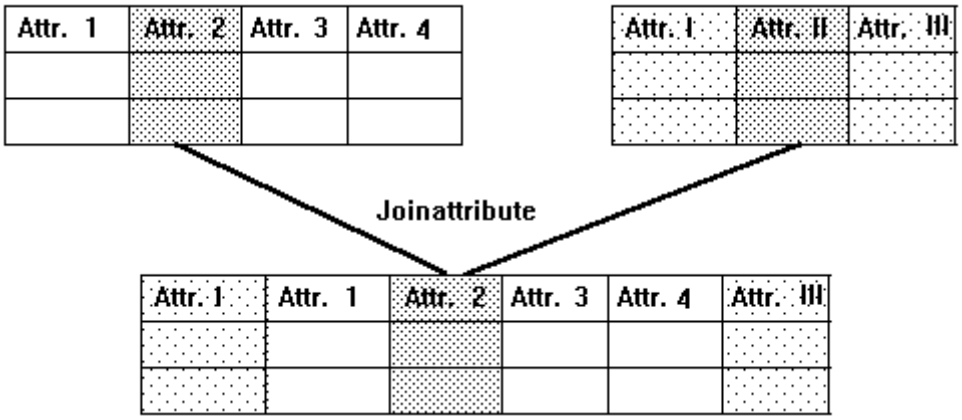


Abbildung 16: Join

Ein allgemeiner Join über den Relationen *R* und *S* ist wie folgt definiert:

$$R[A \otimes B]S := \{r \cup s \mid r \in R \wedge s \in S \wedge [A \otimes B] = true\} \quad mit \quad \otimes = \{=, <, >\leq, \geq, <>\}$$

Dabei steht *A* für ein beliebiges Attribut aus *R* und *B* für ein beliebiges Attribut aus *S*. Mehrere Join-Bedingungen können mittels logischer Operatoren verknüpft werden.

In der relationalen Algebra existieren neben dem allgemeinen Join noch weitere Formen von Join-Operationen.

- Beim *Equi-Join* werden die Tupel aus *R* und *S* miteinander verbunden, bei denen die Vergleichsbedingung *A=B* erfüllt ist.
- Beim *Natural Join* werden die Attribute, die in *R* und *S* enthalten sind, nur einmal in die Ergebnisrelation übernommen.
- Beim *Outer-Join* werden in die Ergebnisrelation auch die Tupel der linken (left outer join) bzw. der rechten (right outer join) Relation mit aufgenommen, die keinen Join-Partner finden. Dabei wird mit Nullwerten aufgefüllt. Die Kombination aus Left- und Right-Outer-Join wird als Outer-Join oder *Full-Outer-Join* bezeichnet.

2.4 Objektorientiertes Datenmodell

Herkömmliche Datenbanksysteme arbeiten satzorientiert (netzwerkartiges und hierarchisches Datenbankmodell) oder mengenorientiert (relationales Datenbankmodell). Die Datenbasis eines objektorientierten Datenbanksystems (OODBS) besteht im Gegensatz dazu aus einer Sammlung von Objekten, wobei jedes Objekt einen physischen Gegenstand, ein Konzept, eine Idee usw. repräsentiert.

Die einzelnen Objekte werden über eindeutige und unveränderliche Objektidentifikatoren identifiziert, welche vom System vergeben werden. Datenobjekte können neben den sonst üblichen meist numerischen oder alphanumerischen Attributen auch Bestandteile enthalten, die selbst wieder Objekte (also strukturiert) sind. Sie werden deshalb auch als komplexe Objekte bezeichnet.

Es existieren weiterhin Operatoren zum Umgang mit solchen Objekten. Während die logisch zusammengehörigen Daten in relationalen Datenbanken oft über mehrere Tabellen „verstreut“ werden, können sie in einem OODBS in einem Datenbankobjekt gehalten werden. Sollen solche logisch zusammengehörenden Daten aus relationalen Datenbanken abgerufen werden, so müssen sie oftmals aus verschiedenen Tabellen mit Hilfe von vergleichsweise aufwendigen Verbundoperationen „zusammengesucht“ werden. Im OODBS betrifft eine solche Anfrage nur ein Datenobjekt. Da bei komplexen Umweltausschnitten Informationen über bestimmte Objekte in relationalen Datenbanken über sehr viele Tabellen verteilt sein können, können sich daraus erhebliche Leistungsvorteile für OODBS ergeben. Die Struktur komplexer Objekte kann also bezüglich der Unterstützung der auf sie angewendeten Operationen optimiert werden. Genau diese Optimierung widerspricht allerdings dem Prinzip der Datenunabhängigkeit und erweist sich im praktischen Einsatz als u.U. großes Problem. Die auf die Objekte anzuwendenden Operationen sind oftmals unterschiedlich und benötigen unterschiedliche „optimale“ Strukturen.

Ein Vorteil von OODBS liegt darin, dass wesentlich mehr an Semantik innerhalb der Datenbank festgehalten werden kann als bei den traditionellen Datenmodellen. Durch die Verwendung von OODB-Systemen treten keine Paradigmenwechsel bei der Speicherung von Daten mehr auf, wenn Software objektorientiert entwickelt wird. Im Jahr 1989 wurden im sog. ODBMS-Manifesto Kriterien festgelegt, die ein Datenbank-Management-System mindestens erfüllen muss, um als objektorientiertes Datenbank-Management-System zu gelten. Weiterhin werden Kriterien angegeben, deren Erfüllung über die der zwingenden Kriterien hinaus wünschenswert ist. Die folgende Aufzählung basiert auf den Ausführungen in [SST97].

Zwingende Kriterien:

- Verwaltung *komplexer Objekte*
- Sicherstellung der *Objektidentität*
- *Kapselung*
- *Typisierung* von Objekten und Zuordnung zu *Klassen*
- *Klassen-* und *Typhierarchie*
- *Überschreiben*, *Überladen* und *spätes Binden* von Methoden (Spezialisierung)
- *berechnungsvollständige Datenbankprogrammiersprache*
- *Erweiterbarkeit* des DBMS um neue Typen (z.B. für Multimediaanwendungen)
- dauerhafte Speicherung von Objekten (*Persistenz*)
- *Sekundärspeicherverwaltung*
- Synchronisation und Recovery von *Transaktionen*
- Bereitstellung von *Anfragesprachen* (analog SQL)

Optionale Kriterien:

- *Mehrfachvererbung*
- statische *Typprüfung* zur Übersetzungszeit
- Unterstützung *verteilter Datenhaltung*
- Unterstützung von *Entwurfstransaktionen*
- *Versionsverwaltung* für Objekte

In der Folgezeit ergaben sich noch **weitere Kriterien**, die so nicht im ODBMS-Manifesto aufgeführt sind.

- Definition und Prüfung von *Integritätsbedingungen*
- Unterstützung von anwendungsspezifischen *Sichten*
- Unterstützung der *Schemaevolution*
- Realisierung einer *Zugriffskontrolle*

2.5 Objektrelationale Systeme

Bislang ist es für rein objektorientierte Datenbank-Management-Systeme kaum gelungen, diese am Markt zu etablieren. Einige größere Anbieter von DBMS-Produkten (z.B. Oracle, Informix – jetzt ein Produkt der IBM) nahmen in den 1990-er Jahren Erweiterungen in Richtung Objektorientierung an ihren bis dahin rein relational ausgerichteten Produkten vor. So wird beispielsweise mit der Technologie der *DataBlades* von Informix eine Möglichkeit zur Erweiterung des DBMS um neue Typen bereitgestellt. Ähnliche Technologien existieren auch für Oracle und DB2. Die erweiterten Produkte sind auch in der Lage, zumindest auf konzeptueller Ebene, Tabellen zu schachteln oder getypte, strukturierte Attribute zu verwalten. Auch in die SQL-Norm wurden mittlerweile Konzepte der Objektorientierung aufgenommen. Objektrelationale DBMS sollen die Vorzüge relationaler Systeme mit denen der Objektorientierung verbinden.

Die zwischen dem relationalen und dem objektorientierten Datenmodell vorhandenen Entsprechungen (z.B. entspricht eine Entität einem Objekt und der Entitätstyp einer Klasse) bilden die Basis für die Entwicklung von objektrelationalen DBMS. Meist werden Spracherweiterungen um objektorientierte Methoden und Datentypen zur Verbesserung des relationalen Modells und damit zur Handhabung komplex strukturierter Daten vorgenommen. In einigen Lösungsansätzen wird über die Komponenten eines relationalen DBMS eine objektorientierte Zugriffsschicht gelegt. Dies ist mit vergleichsweise wenigen Eingriffen in die bestehenden Systeme zu realisieren und gefährdet die Stabilität der Systeme weniger als Eingriffe in mehrere und tiefer liegende Schichten.

3 Sprachschnittstellen zu DBMS

Nach dem ANSI/SPARC Vorschlag ist der Zugriff auf die Daten einer Datenbank auf der externen, konzeptuellen und/oder internen Ebene möglich. Dabei wird vor allem bei den sprachlichen Schnittstellen auf die Nutzerklasse eingegangen. Die Zugriffe auf den verschiedenen Ebenen erfolgen meist durch verschiedene Benutzerklassen. Die Einteilung der Nutzerklassen erfolgt dabei auch nach dem Grad der Kenntnisse, die die Mitglieder der Klasse über Datenbanken im Allgemeinen und im Speziellen über das Datenbanksystem haben, mit dem sie arbeiten.

Nutzerklassen können sein:

- Endnutzer
- Anwendungsprogrammierer
- Datenbankadministratoren

Die *Endnutze*r arbeiten zumeist mit eigens auf den Anwendungszweck zugeschnittenen Oberflächen oder Anwendungsprogrammen. Ein wesentliches Merkmal dieser Benutzerklasse ist die nahezu ausschließliche Arbeit mit Datenbankinhalten. Deshalb werden von den Endnutzern i.d.R. Datenmanipulationsanweisungen (die zur Datenmanipulationssprache – DML gehören) verwendet.

Anwendungsprogrammierer nutzen i.d.R. ein breites Spektrum an Befehlen und Funktionen, meist in Form von speziellen Programmiersprachen (Sprachen der 4. Generation – 4GL) bzw. Einbettungen von Datenmanipulations- bzw. Datendefinitionsanweisungen in andere Programmiersprachen (Trägersprachen/ Wirtssprachen). Für solche Einbettungen stehen verschiedene Ansätze zur Verfügung.

- Bei einem Ansatz wird die Trägersprache um eine Möglichkeit ergänzt, Datenbankanweisungen in den Programmen zu verwendet (z.B. embedded SQL). Die Datenbankanweisungen müssen i.d.R. gekennzeichnet werden (z.B. über `EXEC SQL` am Zeilenanfang). Bevor das Programm vom Compiler der Trägersprache verarbeitet werden kann, wird der Quelltext von einem sog. Präprozessor/ Precompiler verarbeitet. Dieser überträgt die eingebetteten Datenbankanweisungen in Anweisungen und Funktionsaufrufe der Trägersprache.
- Bei einem anderen Ansatz (z.B. ODBC, JDBC) erfolgt der Zugriff auf die Datenbanken über Funktionsaufrufe, denen Datenbankanweisungen als Parameter übergeben werden und die als Funktionsergebnisse bspw. Treffermengen liefern. Spezielle Daten- bzw. Objekttypen, die ebenfalls mit der Schnittstelle zur Verfügung gestellt werden (z.B. Result Sets oder Record Sets), ermöglichen die Verarbeitung der Ergebnisse von Anfragen. Der Vorteil dieses Ansatzes liegt vor allem darin, dass kein zusätzlicher Präprozessor zur Verarbeitung des Quelltexts notwendig ist.

In den Aufgabenbereich von Datenbankadministratoren fällt auch die Verwaltung der internen Speicherungsstrukturen von Datenbanken. Für diese Zwecke stehen Datendefinitionsanweisungen (die Datendefinitionssprache – DDL) zu Verfügung.

Die Anweisungen der Datendefinitionssprache und der Datenmanipulationssprache werden i.d.R. auf einander abgestimmt und liegen meist nicht getrennt voneinander vor.

3.1 Grundlagen von Datenmanipulationssprachen und Abfragesprachen

Datenmanipulationssprachen sind meist vom verwendeten Datenmodell abhängig. Sie dienen zur Formulierung von Anfragen, zum Einfügen, Ändern und Löschen von Daten.

Unterschieden wird hier in navigierende und deskriptive Sprachen. Bei navigierenden Sprachen ist anzugeben, wie auf eine Menge von Daten in einer Datenbank zugegriffen werden soll. Navigierende Sprachen erlauben das "Stöbern" im Datenbestand. Das heißt, die Suche nach Datensätzen kann bspw. auch nach dem Auffinden einer hinreichenden Datenmenge abgebrochen werden. Navigierende Sprachen sind i.d.R in höhere Programmiersprachen eingebettet und bspw. als Schnittstelle bei hierarchischen oder netzwerkorientierten Systemen zu finden.

Bei deskriptiven Sprachen werden Kriterien angegeben, die die gesuchten Daten erfüllen müssen. Bei deskriptiven Sprachen müssen die Benutzer die Eigenschaften der gesuchten Daten kennen und angeben. Das Ergebnis einer Anfrage ist eine Liste aller Daten (*Treffermenge*), die die geforderten Eigenschaften besitzen. Zur Beschreibung der Kriterien, die Elemente einer Treffermenge erfüllen müssen, sind zwei Ansätze gebräuchlich, die im Folgenden kurz beschrieben werden.

3.1.1 Relationenalgebra-Sprachen

<FWB> Algebra : Lehre von Beziehungen zwischen mathematischen Größen

Ein algebraisches System besteht aus einer nicht leeren Menge von Elementen (Trägermenge) und aus einer Menge von Operationen über diese Trägermenge. Die Ergebnisse der Operationen sind wieder Elemente der Trägermenge.

Bei der Relationenalgebra stellen die Relationen die Trägermenge dar. Die Menge der Operationen beinhaltet die klassischen Mengenoperationen Durchschnitt, Vereinigung und Differenz, sowie die Operationen über Relationen, Projektion, Selektion und Join.

Die Verwendung des algebraischen Ansatzes soll anhand des folgenden Beispiels kurz dargestellt werden. Gegeben sei eine Relation `MITARBEITER`, die die Attribute `persnr`, `name`, `vorname`, `plz`, `wohnort`, `strasse`, `brutto` und `imbetriebseit` enthält. Es sollen die Namen, Vornamen und Personalnummern aller Mitarbeiter gesucht werden, die bereits seit 1992 oder davor im Betrieb sind und deren Bruttoverdienst unter 1000 € liegt.

Die Aufgabenstellung beinhaltet zwei verschiedene Operationen, die auf die Tabelle `MITARBEITER` anzuwenden sind. Zunächst müssen die Daten der Mitarbeiter *selektiert* werden, die seit mindestens 1992 im Betrieb beschäftigt sind und weniger als 1000 € verdienen. Auf das Ergebnis dieser Selektionsoperation ist noch eine

Projektionsoperation anzuwenden, die lediglich Personalnummern, Namen und Vornamen im Ergebnis belässt.

In der in *Abschnitt 2.3* angegebenen Notation zur Relationenalgebra könnte die entsprechende Abfrage wie folgt angegeben werden:

```
MITARBEITER[imbetriebseit <=1992 & brutto < 1000][persnr, name, vorname]
```

Eine DML, die auf der Relationenalgebra basiert, ist die Sprache ISBL (Information System Base Language). ISBL kennt u.a. die in der folgenden Tabelle aufgelisteten Operatoren:

Operation	Operatorzeichen
Projektion	%
Selektion	:
natürlicher Join	*
Vereinigung	+
Durchschnitt	.
Differenz	-

Anhand des oben beschriebenen Beispiels soll die Verwendung von ISBL kurz dargestellt werden:

```
LIST (MITARBEITER%persnr,name,vorname) : imbetriebseit<=1992 & brutto<1000
```

3.1.2 Relationenkalkül-Sprachen

<FWB> Kalkül : Rechenoperation, Berechnung, Überlegung

Relationenkalkül-Sprachen beruhen auf der Anwendung des Prädikatenkalküls der 1. Stufe.

$\sim, <, >, \rightarrow, \leftrightarrow, \exists, \forall$

Ein Prädikatenkalkül ist eine formale Sprache, die die korrekten Zeichenreihen zur Darstellung von Eigenschaften definiert. Es werden Regeln eingeführt, mit deren Hilfe von gegebenen Formeln und Zeichenreihen auf andere Formeln und Zeichenreihen geschlossen werden kann.

Beispiele: $\forall\ a,b,c : a < b \wedge b < c \rightarrow a < c$

Für alle a,b,c gilt, wenn a kleiner b und b kleiner c ist, dann muss auch a kleiner c sein.

$$(\exists\ f)\ (f(a) = b \wedge (\forall\ x)(p(x) \Rightarrow f(x) = g(x,f(h(x)))))$$

Es existiert eine Funktion f für die $f(a)=b$ ist und für jedes x, für das p(x) wahr ist, ist dann auch $f(x)=g(x,f((h(x)))$ wahr.

Ein Ausdruck des Relationenkalküls wird deskriptiv angewendet. Eine allgemeine Form eines solchen Kalkül-Ausdrucks kann folgendes Aussehen haben:

```
{ zielliste : selektionsprädikat }
```

Sowohl in der Zielliste als auch im Selektionsprädikat werden Variablen verwendet, die vom Typ *Tupel einer bestimmten Relation* sind. Unter bestimmten Umständen, zum Beispiel beim "Join auf sich selbst", wird die Formulierung des Kalkül-Ausdrucks erst durch die Verwendung der Variablen möglich. Die Definition einer Variablen erfolgt beispielsweise in der Form:

```
RANGE OF <variable> IS <relation>
```

Die einzelnen Attributnamen werden dann in einer qualifizierten Form angegeben. Diese hat folgendes allgemeines Aussehen:

```
<variable>.<attributname>
```

Weitere Elemente des Relationen-Kalküls sind:

- Vergleichsterme:
  - monadisch `<Variable> <Vergleichsoperator> <Konstante>`
  - dyadisch `<Variable> <Vergleichsoperator> <Variable>`
- Bereichsterme: `<Variable> IN <Relation>`
- Junktoren : `NOT, AND, OR`
- Quantoren : ∀, ∃
- Klammern : `() , {}`

Eine Realisierung der Abfrage des Beispiels aus *Abschnitt 3.1.1* könnte mit Hilfe des Relationenkalküls folgendes Aussehen haben:

```
RANGE OF m IS mitarbeiter
{(m.persnr,m.name,m.vorname):m.imbetriebseit<=1992 AND m.brutto<1000}
```

Als Beispiel für eine DML, die auf dem Relationenkalkül basiert, kann QUEL (Query Language) des DBMS INGRES genannt werden. Natürlich wurde beim Entwurf von QUEL darauf geachtet, die Sprache auch für den "normalen Menschen" anwendbar zu gestalten. So wird eine Abfrageoperation mit dem Schlüsselwort `RETRIEVE` eingeleitet. Das Prädikat werden mit dem Schlüsselwort `WHERE` eingeleitet.

Eine Realisierung der Abfrage des Beispiels könnte mit Hilfe von QUEL folgendes Aussehen haben:

```
RANGE OF m IS mitarbeiter
RETRIEVE (m.persnr,m.name,m.vorname)
WHERE m.imbetriebseit<=1992 AND m.brutto<1000
```

3.1.3 Abbildungsorientierte Sprachen

Ziel von abbildungsorientierten Sprachen ist die Verknüpfung von Relationenalgebra und Relationenkalkül. Sie kommt der Denkweise der Datenbanknutzer sehr entgegen und lehnt sich weitestmöglich an natürliche Sprachen an. Die bedeutendste und heute am häufigsten vorkommende Datenmanipulationssprache ist SQL (1974 von Chamberlin und Boyce als **S**E**Q**UE**L** / **S**tructured English **Q**uerry **L**anguage vorgestellt). Sie gilt heute auch als Standard.

Eine Realisierung der Abfrage des Beispiels aus *Abschnitt 3.1.1* könnte mit Hilfe des Relationenkalküls folgendes Aussehen haben:

```
SELECT persnr, name, vorname FROM mitarbeiter
WHERE imbetriebseit<=1992 AND brutto<1000
```

3.1.4 Grafikorientierte Sprachen

Ausgangspunkt bei grafikorientierten Sprachen ist ein Tabellenskelett oder eine Abfragemaske, in welche die Auswahlkriterien eingetragen werden.

Als Vertreter kann die Mitte der 70-er Jahre von Zloof entwickelte Sprache QBE (Query by Example) genannt werden. Die meisten, der derzeit verfügbaren DBMS-Produkte bieten grafikorientierte Abfrage- und Manipulationsmöglichkeiten an. Ein bekanntes Beispiel ist sicherlich MS Access. Die beiden folgenden Abbildungen zeigen einfache Möglichkeiten, die Beispielabfrage mit MS Access zu formulieren.

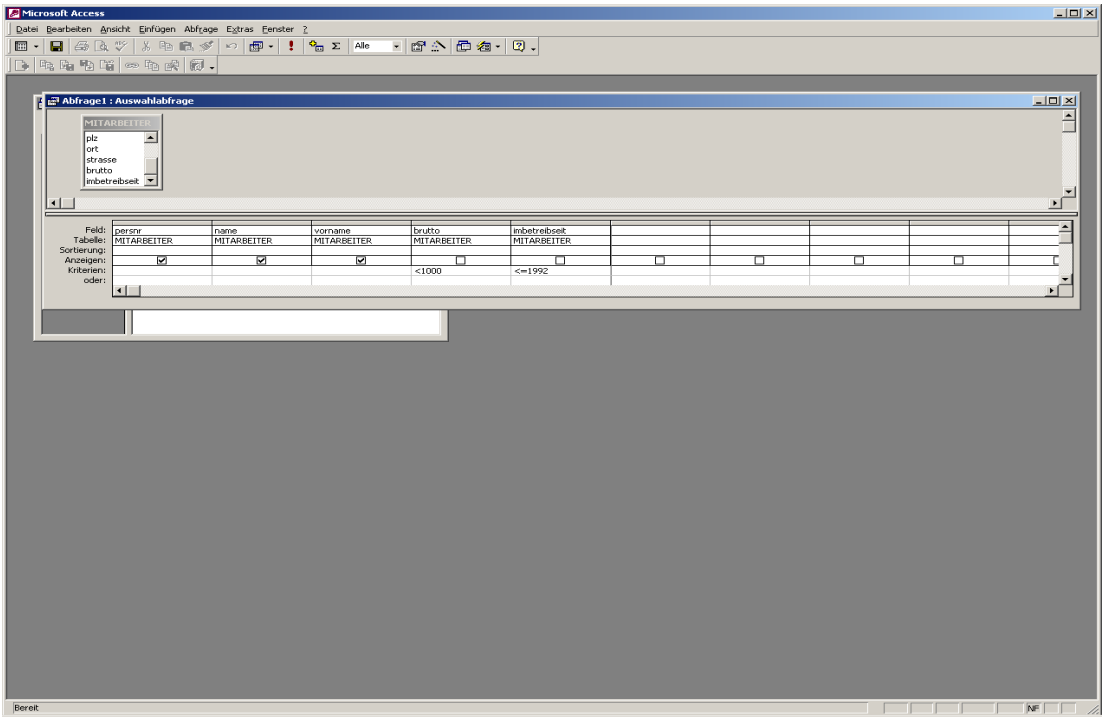


Abbildung 17: Abfrageerstellung unter MS Access

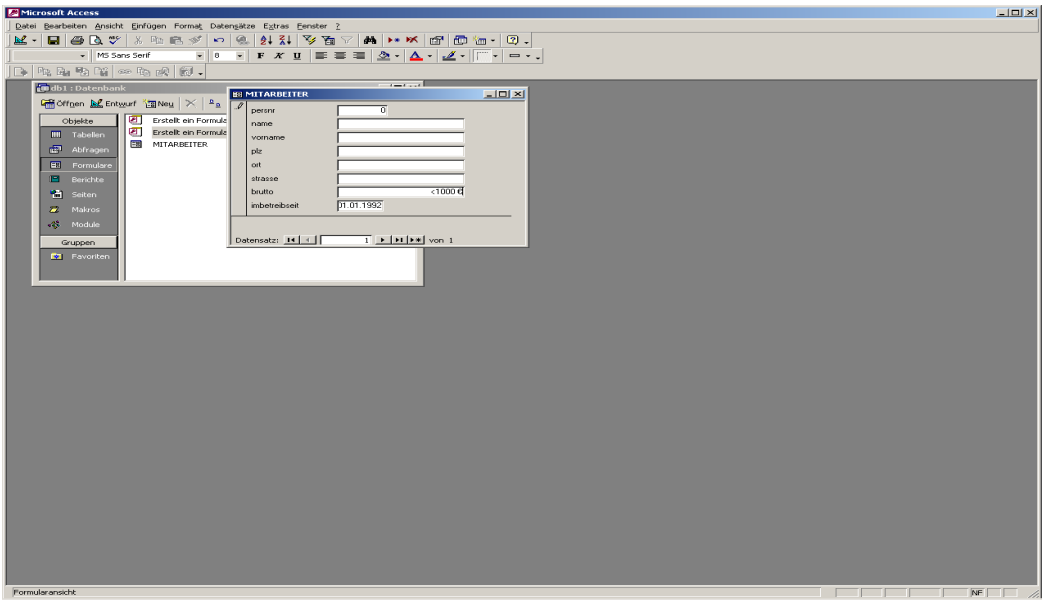


Abbildung 18: Access-Formular

3.2 Kurzer Überblick über SQL

Die Structured Query Language (SQL) ist zwar weitestgehend als unabhängiger Standard akzeptiert, der Sprachumfang ist aber von der jeweiligen Implementierung des DBMS abhängig. Der genaue Sprachumfang muss deshalb aus der Dokumentation des jeweiligen DBMS-Produkts entnommen werden. SQL enthält u.a. Datendefinitions- und Datenmanipulationsanweisungen, von denen die Wichtigsten hier kurz vorgestellt werden sollen. Eine kompakte Darstellung findet sich auch in [Kud07].

Datendefinitionsanweisungen dienen dem Verwalten von Strukturelementen einer Datenbank (Tabellen, Indexe usw.).

- `CREATE` dient zum Erstellen von neuen Strukturelementen
- `ALTER` dient zum Verändern der (statischen) Eigenschaften von Strukturelementen
- `DROP` dient zum Löschen von Strukturelementen

Zu den *Datenmanipulationsanweisungen* zählen Anweisungen zum Durchsuchen des Datenbestands (Retrieval-Anweisungen) und Anweisungen zum Verändern von Datenbankinhalten (Update-Anweisungen).

- Mit Hilfe der `SELECT`-Anweisung können Datenbestände durchsucht werden.
- `INSERT` dient zum Einfügen von Tupeln in eine Tabelle.
- `DELETE` dient zum Löschen von Tupeln aus einer Tabelle.
- `UPDATE` ermöglicht das Ändern von Attributwerten von Tupeln.

Datendefinition

```
CREATE TABLE  <basistabelle>
( <spaltenname> <datentyp> [ NOT NULL [ UNIQUE]],...,
[ UNIQUE (<spaltenname>,... )] )
```

NOT NULL	spezifiziert ein "Mussfeld", d.h. in allen Zeilen der Tabelle muss dieses Feld mit einem Wert versorgt werden
UNIQUE	garantiert eindeutige Attributwerte in den spezifizierten Spalten
PRIMARY KEY	Definition eines Primärschlüssels
FOREIGN KEY	Definition von Fremdschlüsseln
CHECK	Vorgabe einfacher Integritätsbedingungen

Beispiel zu CHECK :

```
CREATE TABLE  gehalt
(    persnr  ....
     name    ....
     gehalt   ...
     CHECK (  gehalt < 3000 ) )
```

Relationale DBMS können die Einhaltung referenzieller Integrität überwachen und sicherstellen. So kann z.B. vor dem Eintragen einer Kundennummer in eine Tabelle AUFTRAG vom System die Prüfung erzwungen werden, ob die Kundennummer tatsächlich im Kundenstamm existiert. Existiert sie nicht, so wird die Änderung am Datenbestand verweigert.

Beispiel:

```
CREATE TABLE AUFTRAG
(   AUFNR ...
    AKNDNR  ...
    AWARNR  ...
    AMENGE  ...
    PRIMARY KEY  (AUFNR),
    FOREIGN KEY   (AKNDNR)  REFERENCES  KUNDE (KDNR),
    FOREIGN KEY   (AWARNR)  REFERENCES  WARE )
```

Datenmanipulation: Änderungsanweisungen

Es existieren drei mengenorientierte Anweisungen:

```
DELETE   FROM <tabelle> [ WHERE <bedingung> ]

UPDATE <tabelle> SET <spalte>=<ausdruck>,...[WHERE <bedingung>]

INSERT  INTO  <tabelle>  {[(<spalte>,...)]  VALUES  (<wert>,...)  |
<SELECT-Anweisung>}
```

Alle NOT NULL - Felder müssen bei INSERT mit Werten „versorgt“ werden! Bei einem Mengen-INSERT ist das Übertragen von Datenmengen, die bspw. mit einer SELECT-Anweisung erzeugt wurden, in eine Tabelle möglich.

Datenmanipulation: SELECT-Anweisung

Die SELECT- Anweisung kann auftreten

- als eigenständige Anweisung
- oder in anderen SQL- Anweisungen wie z.B. INSERT, CREATE VIEW, DECLARE CURSOR.

Als Ergebnis wird eine Tabelle erzeugt.

Format der SELECT-Anweisung:

`SELECT <ergebnisspalten>` ←	SPALTENAUSWAHL " Projektion "
`FROM <tabelle> [<tabalias>], ...` ←	TABELLENAUSWAHL
`[WHERE <bedingung>` ←	ZEILENAUSWAHL " Selektion "
`[GROUP BY <spalte>, ...]` `[HAVING <bedingung> ]` ←	GRUPPIERUNG
`[ORDER BY {<spalte>|<ganzzahl>}` `{ ASC|DESC}] ]` ←	SORTIERUNG

Auswahl der Ergebnisspalten:

```
<ergebnisspalten>:= {ALL | DISTINCT} {{*|<spalte>|<ausdruck>},...}
```

Zeilenauswahl (WHERE- Klausel):

```
                  ⎧  =  ⎫
                  ⎪  <  ⎪
 <ausdruck> =     ⎨  >  ⎬      <ausdruck>
                  ⎪  <= ⎪
                  ⎪  >= ⎪
                  ⎩  <> ⎭

                  ⎧  =  ⎫
                  ⎪  <  ⎪  ⎧ ANY  ⎫
 <ausdruck> =     ⎨  >  ⎬  ⎨ SOME ⎬    ( <select> )
                  ⎪  <= ⎪  ⎩ ALL  ⎭
                  ⎪  >= ⎪
                  ⎩  <> ⎭
```

```
<ausdruck1> [NOT] BETWEEN <ausdruck2> AND <ausdruck3>

<ausdruck> [NOT]  IN      ({<select>  |  <konstante>})

<attribut> [NOT]  LIKE  "<konstante>"
```

```
<attribut>  IS [NOT]  NULL
[NOT ] EXISTS  (<select>)
<bedingung>     { AND | OR }    <bedingung>
NOT  <bedingung>
(<bedingung>)
```

Verknüpfung einer Tabelle mit sich selbst:

z.B. Bei der Frage: "Welche Waren wurden in mehr als einem Auftrag angefordert ?"


```
X.WarenNr=Y.WarenNr
(gleiche Ware)
X.AuftrNr<>Y.AuftrNr
(unterschiedlicher Auftrag)
```

```
SELECT DISTINCT X.WarenNr
FROM AUFTRAG X, AUFTRAG Y
WHERE X.WarenNr=Y.WarenNr
AND X.AuftrNr<>Y.AuftrNr
```

☞ `X.AuftrNr<>Y.AuftrNr` verhindert die Verknüpfung eines Tupels mit sich selbst

Aggregatfunktionen ("Builtin Functions"):

Anzahl:	`COUNT ( {[DISTINCT] <spalte> | * } )`
Summe:	`SUM ( {<ausdruck> |[DISTINCT] <spalte>})`
Durchschnitt:	`AVG ( {<ausdruck> |[DISTINCT] <spalte>})`
Maximum:	`MAX ( {<ausdruck> |[DISTINCT] <spalte>})`
Minimum:	`MIN ( {<ausdruck> |[DISTINCT] <spalte>})`

GROUP BY / HAVING:

`GROUP BY <spalten>` gruppiert konzeptionell die durch FROM, WHERE spezifizierte Datenmenge bezüglich eines Gruppenkennzeichens.

`HAVING` definiert eine Bedingung für die Auswahl von Gruppen ("WHERE-Klausel für Gruppen").

Sortierung der Ergebnisspalten:

```
ORDER BY
{<spalte>|<spaltennummer>}[,{<spalte>|<spaltennr>}...][{ASC|DESC}]
```

Unterabfragen:

Unterabfragen sind in WHERE oder HAVING-Klauseln möglich.

- mit Vergleichsoperatoren – dann muss die Unterabfrage genau einen Wert liefern
- mit `IN`, `SOME`, `ALL`, `ANY` – dann kann die Unterabfrage mehrere (atomare) Werte liefern
- mit `EXISTS` – dann ist die Bedingung erfüllt, sobald die Unterabfrage mindestens einen Wert liefert

View-Konzept

Als View wird eine logische Sicht (auch als ein Fenster vorstellbar) auf bestehende Tabellen bezeichnet. Eine View ist eine "pseudo"- Tabelle, die keine Daten enthält. Lediglich die Beschreibung zum Erzeugen der Sicht wird im Datenbankkatalog gespeichert. Diese Beschreibung ist statisch und von mehreren Benutzern gleichzeitig verwendbar.

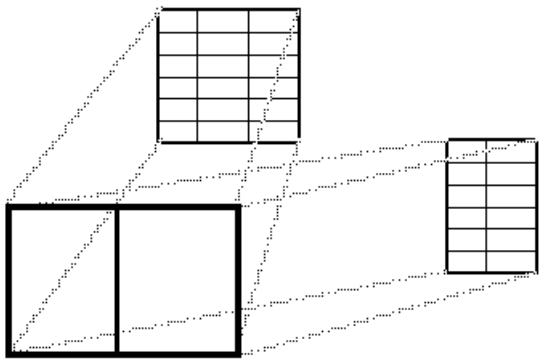


Abbildung 19: View

```
CREATE VIEW <view> [( <spalte>, ... )] AS SELECT ...
[ WITH CHECK OPTION ]
```

Bezüglich der Angabe von `ORDER BY`-Klauseln in der SELECT-Anweisung können Einschränkungen existieren. Es können auch Views definiert werden, die wieder auf Views basieren.

Anwendungsbeispiele für Views:

- Vereinfachung sich wiederholender Abfragen
- Erreichen logischer Datenunabhängigkeit
- Zugriffschutz und Zugriffkontrolle

Wenn Änderungsoperationen auf Daten in Sichten ausgeführt werden sollen, so können verschiedene Einschränkungen existieren, wie z.B.:

- In der Sicht dürfen keine zusammengesetzten oder berechneten Werte enthalten sein.

- Alle Muss-Spalten der entsprechenden Basistabelle müssen in der Sicht enthalten sein.
- `GROUP BY`- oder `DISTINCT`-Klauseln dürfen in der Sichtdefinition nicht enthalten sein.
- Die Sicht darf sich nur über eine Tabelle erstrecken.

Datenschutz/Zugriffskontrolle

Voraussetzungen dafür sind, dass eine Benutzeridentifikation durch das System ermöglicht wird und der Ersteller der Basistabellen oder Views alle Zugriffsrechte auf das Objekt jeweilige besitzt und anderen Benutzern abgestufte Zugriffsrechte erteilen kann.

- Mit dem View-Konzept kann der Zugriff von Benutzern auf Tupel-Ebene beschränkt werden.
- Mit der Anweisung `GRANT` kann der Datenzugriff bis auf Spaltenebene geregelt werden. Weiterhin können Benutzern Rechte zum Ausführen von Anweisungen gewährt werden.
- Das Gegenstück zur `GRANT`-Anweisung ist die Anweisung `REVOKE`, mit der Benutzern Rechte entzogen werden können.

Vergabe von Rechten bezüglich der Ausführung von Anweisungen:

```
GRANT {<anweisungen>|ALL} TO {<benutzer>|<rolle>|PUBLIC}
[WITH GRANT OPTION]
```

Vergabe von Zugriffsrechten auf Daten:

```
GRANT {select|update|insert|delete|ALL} [(<spalten>)] ON <objekt> TO
{<benutzer>|<rolle>|PUBLIC} [WITH GRANT OPTION]
```

Entziehen von Rechten bezüglich der Ausführung von Anweisungen:

```
REVOKE {<anweisungen>|ALL} FROM {<benutzer>|<rolle>|PUBLIC}
```

Entziehen von Zugriffsrechten auf Daten:

```
REVOKE {select|update|insert|delete|ALL} ON <objekt> FROM
{<benutzer>|<rolle>|PUBLIC} [WITH GRANT OPTION]on
```

Transaktionssteuerung

Ob Transaktionen, die aus mehreren SQL-Anweisungen bestehen, mit Anweisungen zur Transaktionssteuerung eingeleitet und beendet werden müssen oder ob es ausreichend ist, Transaktionen nur abzuschließen oder zurückzusetzen und ob Transaktionen, die nur aus einer SQL-Anweisung bestehen, explizit abgeschlossen

werden müssen, hängt vom vom jeweiligen DBMS verwendeten Transaktionskonzept ab.

- Start einer „Transaktionsklammer“

```
{START TRANSACTION | BEGIN [{WORK | TRANSACTION}]}
```

- Transaktion (erfolgreich) abschließen und evtl. Abbau der Verbindung zum DBS

```
COMMIT [{WORK | TRANSACTION}] [RELEASE]
```

- Transaktion zurücksetzen

```
ROLLBACK [{WORK | TRANSACTION}] [TO SAVEPOINT <sicherungspunkt>]
```

- Sicherungspunkt innerhalb einer Transaktion setzen

```
SAVEPOINT <sicherungspunkt>
```

- Sicherungspunkt löschen

```
RELEASE SAVEPOINT <sicherungspunkt>
```

- Transaktionseigenschaften festlegen

```
SET TRANSACTION [READ ONLY | READ WRITE] [ISOLATION LEVEL
{READ COMMITTED | READ UNCOMMITTED |REPEATABLE READ |SERIALIZABLE}]
```

4 Datenbankentwurf

4.1 Überblick über den Datenbankentwurfsprozess

4.1.1 Anforderungen

Der Entwurf eines Datenbankschemas für einen komplexen Umweltausschnitt ist ein aufwendiger und tw. schwieriger Prozess, da in den meisten Fällen bei *n* verschiedenen Nutzern auch *n* verschiede Sichten auf die zu entwerfende Datenbank existieren. Da die zu entwerfende Datenbank die Grundlage für alle höheren Schichten (Anwendungsserver, Anwendungsprogramme, ...) ist, muss der Datenbankentwurf mit großer Sorgfalt vorgenommen werden, da sich Fehler später oft nur mit hohem Aufwand beseitigen lassen.

Um also ein Datenbankschema zu entwickeln, dass allen Anforderungen möglichst gerecht wird, sollten folgende Anforderungen an den Entwurfsprozess berücksichtigt werden:

- Anwenderorientiertheit
    - Der Entwurfsprozess sollte unter Beteiligung der Anwender stattfinden.
    - Das Datenmodell sollte in der Begriffswelt der Anwender formuliert sein.
- Determiniertheit
    - Für eine gegebene Problemstellung und einen Umweltausschnitt darf es nur genau ein Datenmodell geben.
    - Unabhängig von der Interpretation muss der Entwurfsprozess genau dieses Datenbankschema ergeben.
- Teilbarkeit
    - Beim Entwurf eines Datenbankschemas muss das Gesamtschema in einzelne Teilschemata aufteilbar sein.
- Integrierbarkeit
    - Datenbankschemata, die nur einen Teil des Umweltausschnittes darstellen, müssen in das Schema, das den gesamten Sachverhalt darstellt, integriert werden können.

Um ein möglichst genaues Abbild der Realität zu erzeugen und somit Voraussetzungen für eine hohe Akzeptanz bei allen betroffenen Nutzern zu erreichen, sind diese Forderungen eine Voraussetzung.

4.1.2 Entwicklung von Datenbankschemata

Für die Entwicklung von datenbankgestützten IT-Lösungen müssen jeweils zwei Entwurfsphasen berücksichtigt werden, ein einmaliger Datenbankentwurfsprozess (DEP) und ein *n*-maliger Softwareentwicklungsprozess (SEP) für die verschiedenen Anwendungen, die auf die (zentrale) Datenbank zugreifen.

Jeder SEP hat dabei in den meisten Fällen eine eigene Sicht auf die Datenbank, die beim Datenbankentwurf berücksichtigt werden muss.

Der DEP bildet mit Hilfe von Daten- und Beschreibungsmodellen einen Umweltausschnitt (Miniwelt) auf ein Schema ab. Im einfachsten Fall muss nur eine Sicht berücksichtigt werden. Dabei wird auch beim Entwurf einer Datenbank der ANSI/SPARC Vorschlag zu Grunde gelegt.

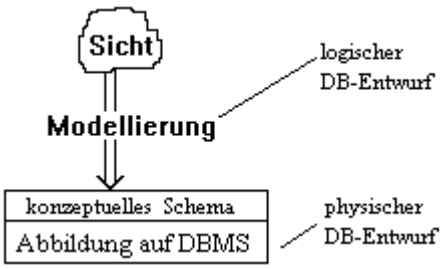


Abbildung 20: Datenbankentwurf für eine Anwendung

Meist müssen aber mehrere Sichten auf einer Datenbank abgebildet werden, wobei die einzelnen Sichten nach ihrer Modellierung zu einem konzeptuellen Schema zusammengefasst werden müssen. Dieses konzeptuelle Informationsschema wird mit Hilfe von Transformationsregeln in ein datenbankspezifisches Schema (vom entsprechenden DBMS verarbeitbares Schema) umgewandelt.

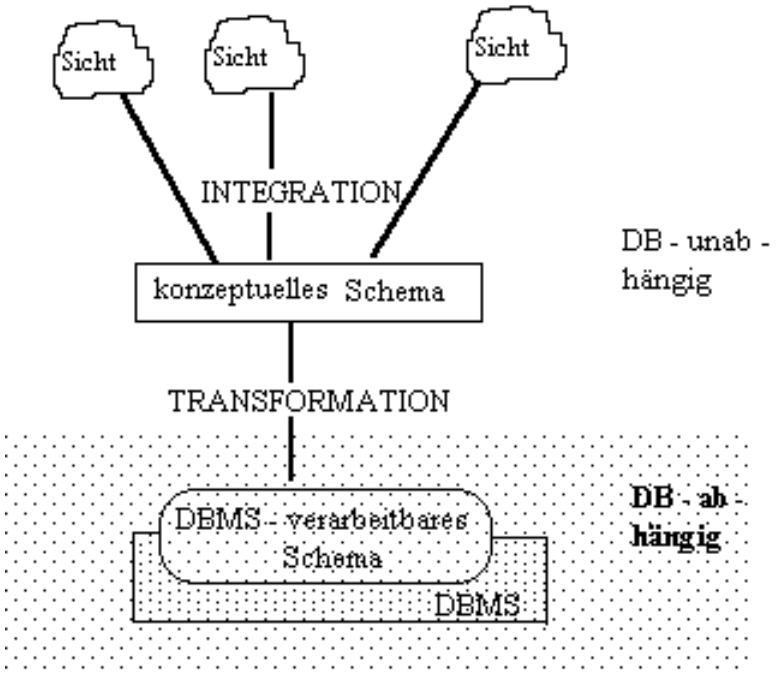


Abbildung 21: Datenbankentwicklung mit mehreren Sichten

4.1.3 Phasen des Entwurfsprozesses

Das in *Abbildung 22* dargestellte Phasenmodell beschreibt den Weg von der Anforderungsanalyse bis hin zum physischen Datenbankentwurf, d.h. bis zur konkreten Implementierung.

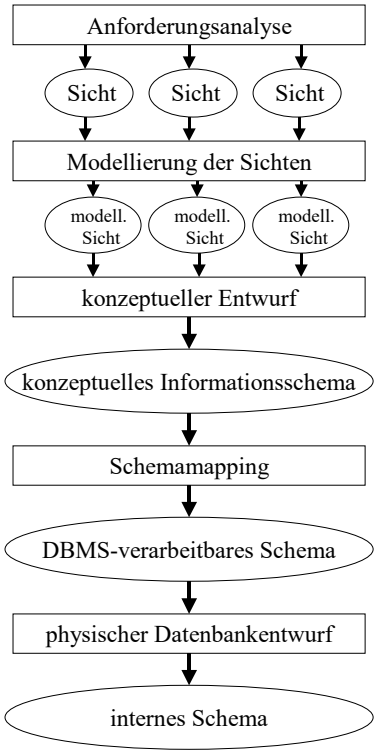


Abbildung 22: Phasenmodell des Datenbankentwurfsprozesses

Grundvoraussetzung für die Erstellung einer Datenbank ist die Anforderungsanalyse, die sich in zwei Analyseprozesse unterteilt, die *informelle Analyse* und die *operationelle (funktionale) Analyse*.

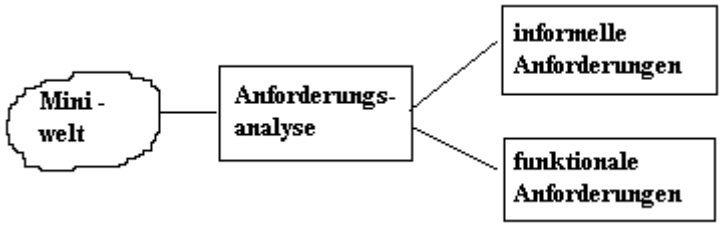


Abbildung 23: Anforderungsanalyse

Beim Entwurf einer Datenbank muss berücksichtigt werden, dass die Miniwelt, die sie darstellt, Änderungen unterworfen ist, die sich auch in der Datenbank niederschlagen müssen, um deren Konsistenz zu gewährleisten. Die Nutzung einer Datenbank samt der notwendigen Aktualisierungen des Datenbankschemas beschreibt der Datenbanklebenszyklus (*Abbildung 24*).

Solche veränderten Anforderungen der Anwendung (bspw. wenn zukünftig zusätzliche Informationen zu berücksichtigen und zu speichern sind) werden unabhängig vom zu Grunde liegenden Datenbanksystem vorgenommen. Hier geht es vorrangig darum, den Informationsbedarf der Anwendung zu befriedigen. Performance-Aspekte spielen eine eher untergeordnete Rolle. Wichtig ist die Wahrung der *logischen* Konsistenz der Datenbank. Weitere Probleme ergeben sich hier bspw. auch auf dem Gebiet der Datenarchivierung, weil sich zu archivierende

Daten auf das jeweilige Datenbankschema beziehen und einmal archivierte Daten aber i.d.R. nicht mehr verändert und an neue Datenbankschemata angepasst werden dürfen. Hier sind Versionierungskonzepte notwendig. Für Änderungen an der logischen Datenorganisation wird oft auch der Begriff *Schemaevolution* verwendet.

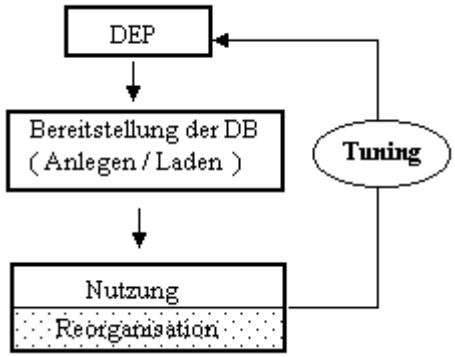


Abbildung 24: Datenbanklebenszyklus

Als *Tuning-Maßnahmen* werde alle Maßnahmen bezeichnet, die das Leistungsverhalten verbessern sollen. Hierzu zählen u.a. Maßnahmen, wie die Neuabstimmung von Systemparametern, die die Performance eines DBMS beeinflussen (z.B. Größe des Puffer-Pools, Parameter zur Steuerung vorausschauenden Lesens usw.), aber bspw. auch die gezielte Denormalisierung von Datenbankschemata.

Unter den Begriff Datenbankreorganisation fallen Maßnahmen, die

- in engem Bezug zum Datenbanksystem stehen,
- den logischen Informationsgehalt der Datenbank nicht verändern und
- üblicherweise zur Verringerung der Speicher- und/oder Verarbeitungskosten

beim Betrieb vorhandener Anwendungen dienen. Dazu zählen bspw.

- Änderungen der internen Struktur einer Datenbank oder
- die Bereinigung bzw. den Neuaufbau interner Speicherungsstrukturen.

4.1.4 Ableitung eines Datenbankschemas aus einer verbalen Spezifikation

Der Entwurfsprozessprozess erfolgt in mehreren Schritten, die nachfolgend beschrieben werden.

Informelle Analyse der Spezifikation

Zunächst wird der in der verbalen Spezifikation beschriebene Umweltausschnitt analysiert. Ziel ist es, die einzelnen Objektklassen und deren Beziehungen untereinander zu erkennen. Je genauer die Spezifikation formuliert ist, desto besser wird es später gelingen, den Umweltausschnitt zu modellieren. Dabei ist es notwendig, die für die Modellierung wichtigen Informationen von den oftmals

vorhandenen Füllworten und Füllsätzen zu trennen. Ist die Spezifikation ungenau, so ist die Qualität des Entwurfes sehr stark von den Fähigkeiten des Datenbank-Designers abhängig. Es ist dann nötig, die fehlenden Informationen zu beschaffen oder die Lücken nach eigenen Erfahrungen auszufüllen.

Funktionale Analyse der Spezifikation:

Hier sind die funktionalen Anforderungen an das System zu spezifizieren. Es wird festgelegt, auf welche Art und Weise die einzelnen Objekte der Objektklassen zu bearbeiten sind. Dieser Schritt wird durchgeführt, wenn der Verwendungszweck der zu speichernden Daten bekannt ist. Die Zusammenhänge zwischen informeller- und funktionaler Analyse zeigt *Abbildung 25*.

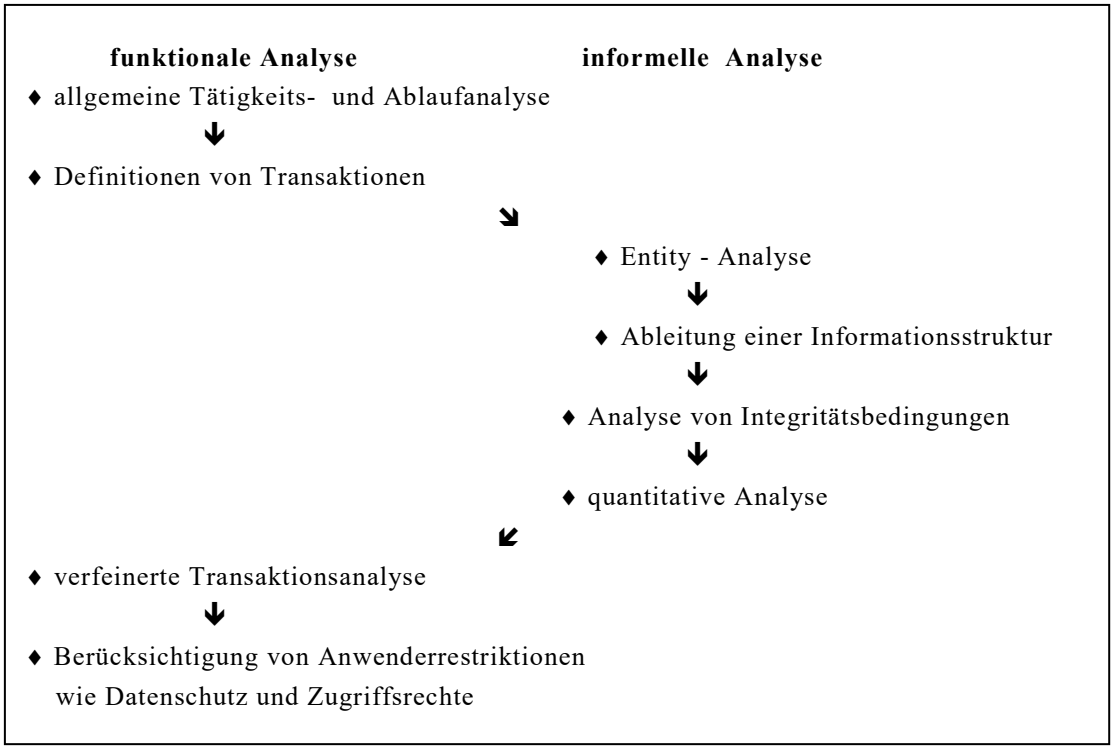


Abbildung 25: Informelle- und funktionale Analyse

Darstellung der Informationsstruktur mit einem semantischen Datenmodell

In der nächsten Phase werden die Ergebnisse der ersten Phase zu einem Modell der Informationsstruktur des Umweltausschnitts zusammengefasst. Dabei werden normalerweise standardisierte und noch vom späteren Datenmodell unabhängige Darstellungsformen verwendet. Eine Darstellungsform, die es ermöglicht, Objektklassen und deren Beziehungen übersichtlich darzustellen, ist das *Entity-Relationship-Modell*.

Ableitung eines dem Zieldatenmodell angepassten Informationsschemas

In dieser Phase wird aus dem erstellten Modell eine Informationsstruktur abgeleitet, welche die Abbildung der Objekte und deren Beziehungen ermöglicht. Dabei werden die zur Abbildung notwendigen Daten in einer Form dargestellt, die von einem Datenbank-Management-System verwaltet werden kann. Viele der heute gebräuchlichen DBMS arbeiten mit dem relationalen Datenmodell.

Physischer Entwurf

Die anschließende Phase beschäftigt sich mit der Anpassung des abgeleiteten (relationalen) Schemas an das konkrete zu verwendende DBMS und die dazugehörige Hardware-Basis. Den einzelnen Attributen werden die bestimmten Datentypen zugeordnet. Hier können auch Integritätsbedingungen und referenzielle Integritäten festgelegt werden. Weiterhin müssen in dieser Phase Forderungen zu Datenschutz und Datensicherheit, wie die Vergabe von Zugriffsrechten für die einzelnen Benutzer, die Benutzerkontrolle usw., berücksichtigt werden. Nach dieser Phase kann die Datenbank auf einem konkreten Rechnersystem implementiert und mit Daten geladen werden.

4.2 Logischer Datenbankentwurf

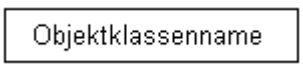
4.2.1 Entity-Relationship-Modell

Das Entity-Relationship-Modell wurde 1976 von Chen [Che76] vorgestellt. Es ist ein unabhängiges Modell zur Darstellung von Objektklassen und deren Beziehungen. Durch seine einfache und übersichtliche grafische Darstellung mittels sogenannter Entity-Relationship-Diagramme (ERD), erreichte dieses Modell eine relativ hohe Verbreitung. Das Entity-Relationship-Modell kennt die nachfolgend beschriebenen Elemente.

Entities (Objekte, Entitäten)

Eine Entität kann unabhängig von anderen existieren und kann eindeutig identifiziert werden (z.B. eine Person, Firma usw.). Eine Darstellung im ERD dient zur Beschreibung einer Menge gleichartiger (*Entity Type*) Entities. Eine solche Menge wird auch als *Entity Set* oder bezeichnet.

Darstellungsmittel:



Relationships (Beziehungen)

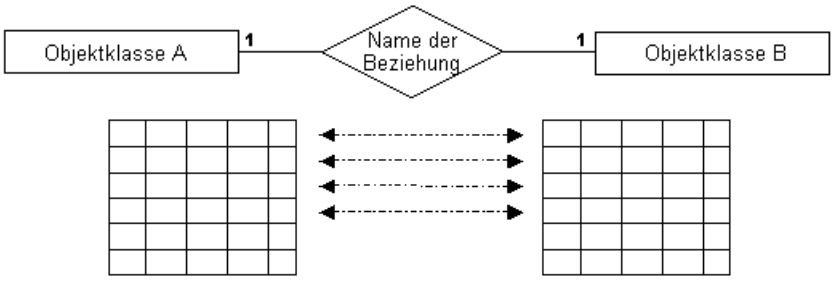
Durch die Beziehungen werden die Verbindungen und Abhängigkeiten zwischen den einzelnen Objekten, meist unterschiedlichen Typs, dargestellt.

Darstellungsmittel:

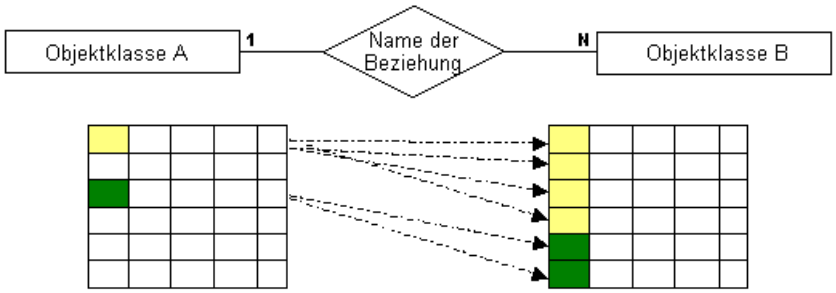
Der Typ einer Beziehung wird durch die Beziehungskardinalität beschrieben. Die Darstellung im ERD dient zur Veranschaulichung eines funktionalen Zusammenhangs zwischen Objekten (normalerweise) unterschiedlicher Entity-Typen. Man bezeichnet dies als *Relationship Type*. Die Kardinalität bezeichnet die Anzahl von Instanzen eines Relationship Type, an denen ein Entity beteiligt sein kann. Ein Relationship Type kann *eindeutig* (**1 : 1**), *einseitig eindeutig* (**1 : N**) oder *komplex* (**M : N**) typisiert werden.

Darstellungsmittel:

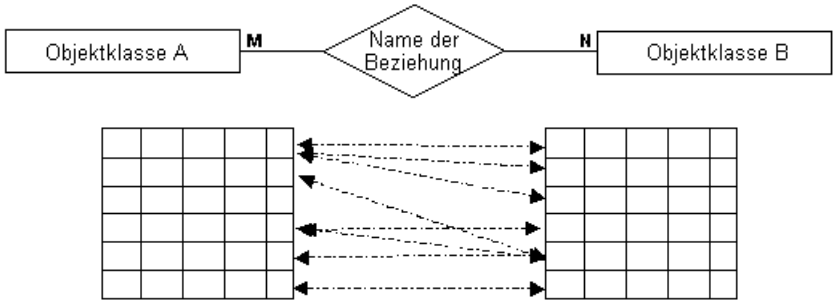
1 : 1- Beziehung (eindeutig)



1 : N- Beziehung (einseitig eindeutig - multiple)



M : N- Beziehung (komplex)



Attribute (Eigenschaften, Merkmale)

Attribute beschreiben die Eigenschaften der einzelnen Objekte oder Beziehungen. Da in größeren Umweltausschnitten normalerweise sehr viele unterschiedliche Objekttypen und Beziehungstypen mit vielen Attributen enthalten sind, wird aus Gründen der Übersichtlichkeit in den grafischen Darstellungen häufig auf die Angabe der Attribute ganz verzichtet oder es werden nur wichtige Attribute (Schlüsselattribute, Attribute zur näheren Beschreibung von Beziehungen) dargestellt.

Darstellungsmittel:

Beispiel:

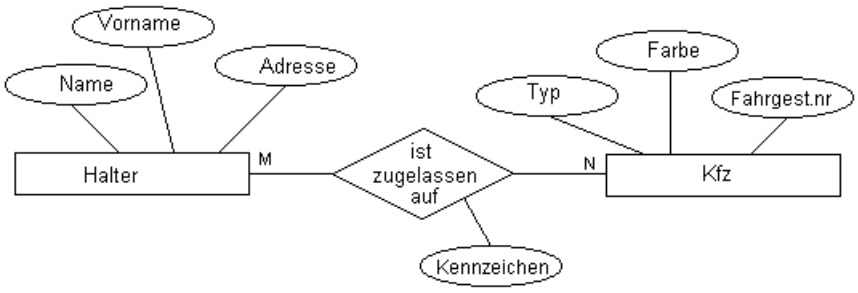


Abbildung 26: Ausschnitt aus einer Datenbank im Kraftfahrzeug-Meldewesen

Für die Ableitung eines relationalen Schemas aus dem ERD gelten die folgenden Regeln:

- Für jede im ERD modellierte Objektklasse ist eine Relation zu bilden, die neben dem Primärschlüssel alle Attribute enthält, die die Objekte der Klasse näher beschreiben.
- Im relationalen Datenmodell werden Beziehungen durch übereinstimmende Werte in Schlüsselfeldern dargestellt. Dazu muss vor allem der Datentyp der Schlüsselfelder gleich sein. In der Regel sind dies auch Felder, die in beiden Tabellen den gleichen Namen besitzen. In den meisten Fällen ist dies der Primärschlüssel aus einer Tabelle, der einen eindeutigen Wert für jeden Datensatz enthält, der mit einem Eintrag in einem Attribut (Fremdschlüssel) der anderen Tabelle übereinstimmt.

Typ	Hersteller	Klasse
V 70	Volvo	mittel
A 4	Audi	mittel

⇧

Primärschlüssel

Fahrzeugtypen

Fahrgest. Nr.	Baujahr	Farbe	Typ
0123	1998	rot	V 70
0432	1996	grün	A 4
0832	1997	blau	A 4
0264	1998	rot	V 70
0326	1997	blau	A 4
0295	1996	gelb	A 4

Autos

⇧

Fremdschlüssel

- Objektklassen, zwischen denen im ERD 1 : 1-Beziehungen auftreten, können normalerweise ohne Informationsverluste zu einer Objektklasse zusammengefasst werden, wenn dem nicht bestimmte Gründe entgegenstehen, wie z.B. Datenschutz oder gewünschte Datenverteilung.

Tabelle AB :

a oder **b**				

- Besteht zwischen zwei Objektklassen eine 1 : N-Beziehung, so kann diese im relationalen Schema dadurch abgebildet werden, dass der Schlüssel der Objektklasse, deren Objekte in Beziehung zu mehreren Objekten der anderen Klasse stehen können, in die Klasse als Fremdschlüssel aufgenommen wird, deren Objekte in Beziehung zu genau einem Objekt der anderen Klasse stehen. Ausgehend von der grafischen Darstellung des ERD wird der Schlüssel der Objektklasse, auf deren Seite die 1 steht, in die Objektklasse als Fremdschlüssel aufgenommen, auf deren Seite das N steht. Eventuell auftretende Attribute zur näheren Beschreibung der Beziehungen, werden in die Objektklasse aufgenommen, die den Fremdschlüssel enthält.

Tabelle A :

a			

Tabelle B:

b	**a**			

- Zwischen zwei oder mehreren Objektklassen auftretende M : N-Beziehungen lassen sich durch das Erzeugen neuer Relationen abbilden. Dabei werden in die Relation die Primärschlüssel der an der Beziehung beteiligten Objektklassen aufgenommen. Der Schlüssel einer solchen Beziehungsrelation setzt sich, wenn kein neuer Schlüssel vergeben werden soll, mindestens aus der Kombination der Fremdschlüssel zusammen. Attribute, die die Beziehung näher beschreiben (z.B. Zeitangaben), werden ebenfalls in die Beziehungs-Relation mit aufgenommen. Häufig ist es auch notwendig, solche Attribute mit in den Schlüssel der Beziehungs-Relation aufzunehmen, um dessen Eindeutigkeit zu gewährleisten.

Tabelle A (a,...)

a		

Tabelle AB (a, b, c ...)

a	**b**	**c**

Tabelle B (b,...)

b		

4.2.2 Beispiel zur Modellierung eines verbal beschriebenen Umweltausschnitts für das relationale Datenmodell

Beschreibung des Umweltausschnitts

Für ein Auskunfts- und Reservierungssystem einer Fachbibliothek ist ein Datenbankschema zu entwerfen. Dabei soll der folgende Umweltausschnitt betrachtet werden.

Jeder Benutzer der Bibliothek kann sich mit Hilfe des Systems Informationen über die einzelnen in der Bibliothek vorhandenen Bücher einholen oder ein Buch reservieren, wenn es derzeit nicht in der Bibliothek vorhanden ist. Die Mitarbeiter der Bibliothek, die ebenfalls zu den Benutzern des Systems zählen, sind darüber hinaus befugt, Bücher zu verleihen und die entsprechenden Leihdaten einzutragen. Von jedem Benutzer werden Nachname, Vorname, Beruf, vollständige Anschrift und Telefonnummer gespeichert. Jedem Benutzer werden bestimmte Nutzungsrechte für das System zugeteilt. Um zu vermeiden, dass eine Kennung von Unbefugten benutzt werden kann, muss sich jeder Benutzer durch seinen Namen und ein ihm bekanntes Passwort ausweisen. Dieses Passwort wird ebenfalls in der Datenbank gespeichert. Weiterhin muss es möglich sein, die Benutzung durch bestimmte Nutzer zeitweilig einzuschränken oder auch Benutzer zu sperren.

Zu den einzelnen Büchern, über die die Bibliothek verfügt, sind ISBN, Titel, Sprache, Auflage, Erscheinungsjahr, Format und die Anzahl Seiten zu speichern. Jedes Buch wird von einem bestimmten Verlag herausgegeben, über den es eventuell auch bezogen werden kann. Deshalb sollen auch Name und Sitz der Verlage festgehalten werden. Weiterhin sollen über die Autoren der Bücher bestimmte Informationen wie Name, Vorname und eine kurze Biografie abrufbar sein. Jedes Buch kann in der Bibliothek in mehreren Exemplaren vorhanden sein. Diese Exemplare können die Benutzer ausleihen. Dabei sollen Ausleihdatum, das Solldatum der Rückgabe, das Datum der tatsächlichen Rückgabe und das Datum der letzten eventuell ergangenen Mahnung gespeichert werden. Um abfragen zu können, ob Exemplare eines bestimmten Buches vorhanden sind, ist für jedes Exemplar festzuhalten, ob und wo es sich in der Bibliothek befindet.

Ist von einem bestimmten Buch derzeit kein Exemplar in der Bibliothek vorhanden, so soll sich der Benutzer für ein Exemplar dieses Buches vormerken lassen können. Sobald ein Exemplar des Buches zurückgegeben wird, wird der Benutzer darüber informiert. Dabei hat der Nutzer keinen Anspruch auf ein bestimmtes Exemplar.

Informelle Analyse der Beispielspezifikation

Diese Phase verlangt vom Datenbank-Designer eine gewisse Übung, um in einer verbal formulierten Spezifikation die einzelnen Objektklassen und deren Beziehungen zu erkennen. Als Beispiel soll der folgende Ausschnitt aus der Spezifikation verwendet werden.

„Zu den einzelnen Büchern, über die die Bibliothek verfügt, sind ISBN, Titel, Sprache, Auflage, Erscheinungsjahr, Format und die Anzahl Seiten zu speichern. Jedes Buch wird von einem bestimmten Verlag herausgegeben, über den es eventuell auch bezogen werden kann. Deshalb sollen auch Name und Sitz der Verlage festgehalten werden."

Aus diesen beiden Sätzen ist zu erkennen, dass am Umweltausschnitt mindestens zwei unterschiedliche Objekttypen beteiligt sind. Es handelt sich hierbei um die Objekttypen **Buch** und der **Verlag**. Zu beiden Objekttypen sind ebenfalls die näher beschreibenden Attribute angegeben. Weiterhin ist zu erkennen, dass zwischen den einzelnen Objekten der Objekttypen eine Beziehung in der Art besteht, *dass ein Buch von einem bestimmten Verlag herausgegeben wird*. Von welchem Typ die Beziehung ist, geht aus der verbalen Beschreibung nicht eindeutig hervor. Der Datenbank-Designer muss diese Informationslücke entweder durch Nachfragen bei einem Experten oder durch seine eigenen Erfahrungen ausfüllen. Es soll hier davon ausgegangen werden, dass ein Buch immer nur von einem Verlag herausgegeben wird.

Wird die gesamte Spezifikation auf diese Art analysiert, so ergeben sich die in der Spezifikation durch Fettdruck gekennzeichneten Objektklassen. Die erkannten Beziehungen sind kursiv gedruckt. Dabei wurden mehrfache Hinweise auf Objektklassen und Beziehungen nur einmal gekennzeichnet.

Es ist für ein Auskunfts- und Reservierungssystem einer Fachbibliothek eine Datenbank zu entwerfen. Dabei soll der folgende Umweltausschnitt betrachtet werden.

Jeder **Benutzer** der Bibliothek kann sich mit Hilfe des Systems Informationen über die einzelnen in der Bibliothek vorhandenen **Bücher** einholen oder ein Buch, wenn es derzeit nicht in der Bibliothek vorhanden ist, *reservieren*. Die Mitarbeiter der Bibliothek, die ebenfalls zu den Benutzern des Systems zählen, sind darüber hinaus befugt, Bücher zu verleihen und die entsprechenden Leihdaten in das System einzutragen. Von jedem Benutzer werden Nachname, Vorname, Beruf, vollständige Anschrift und Telefonnummer gespeichert. Jedem Benutzer werden bestimmte Nutzungsrechte für das System zugeteilt. Um zu vermeiden, dass eine Kennung von Unbefugten benutzt werden kann, muss sich jeder Benutzer durch seinen Namen und ein ihm bekanntes Passwort ausweisen. Dieses Passwort wird ebenfalls in der Datenbank gespeichert. Weiterhin muss es möglich sein, die Benutzung durch bestimmte Nutzer zeitweilig einzuschränken oder auch Benutzer zu sperren.

Zu den einzelnen Büchern, über die die Bibliothek verfügt, sind ISBN, Titel, Sprache, Auflage, Erscheinungsjahr, Format und die Anzahl Seiten zu speichern. Jedes Buch wird von einem bestimmten **Verlag** *herausgegeben*, über den es eventuell auch bezogen werden kann. Deshalb sollen auch Name und Sitz der Verlage festgehalten werden. Weiterhin sollen über die ***Autoren*** *der Bücher* bestimmte Informationen wie Name, Vorname und eine kurze Biografie abrufbar sein. Jedes Buch kann in der Bibliothek in mehreren **Exemplaren** *vorhanden* sein. Diese

Exemplare können die Benutzer *ausleihen*. Dabei sollen Ausleihdatum, das Solldatum der Rückgabe, das Datum der tatsächlichen Rückgabe und das Datum der letzten eventuell ergangenen Mahnung gespeichert werden. Um abfragen zu können, ob Exemplare eines bestimmten Buches vorhanden sind, ist für jedes Exemplar festzuhalten, ob und wo es sich in der Bibliothek befindet.

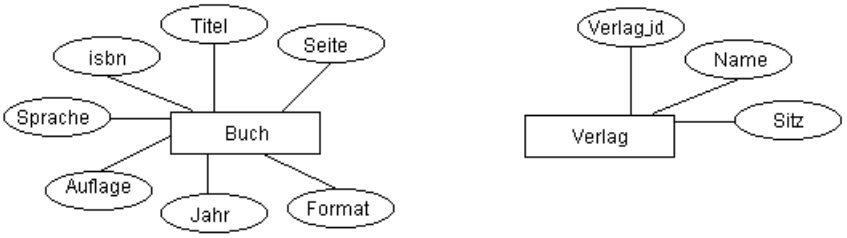
Ist von einem bestimmten Buch derzeit kein Exemplar in der Bibliothek vorhanden, so soll sich der Benutzer für ein Exemplar dieses Buches vormerken lassen können. Sobald ein Exemplar des Buches zurückgegeben wird, wird der Benutzer darüber informiert. Dabei hat der Nutzer keinen Anspruch auf ein bestimmtes Exemplar.

Erstellung eines Entity-Relationship-Diagramms zur Darstellung des beschriebenen Sachverhalts

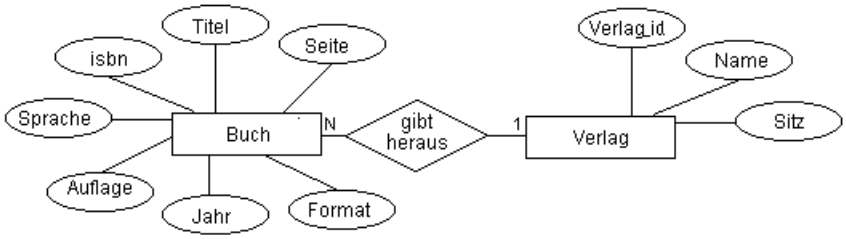
In dieser Phase geht es darum, die Informationen, die bei der informellen Analyse der Spezifikation gewonnen wurden, auf eine möglichst eindeutige, übersichtliche und datenmodellunabhängige Art und Weise darzustellen. Dazu bietet sich ein Entity-Relationship-Diagramm an. Als Beispiel soll der bereits vorher betrachtete Ausschnitt aus der Spezifikation verwendet werden.

„Zu den einzelnen **Büchern,** über die die Bibliothek verfügt, sind ISBN, Titel, Sprache, Auflage, Erscheinungsjahr, Format und die Anzahl Seiten zu speichern. Jedes Buch wird von einem bestimmten **Verlag** *herausgegeben*, über den es eventuell auch bezogen werden kann. Deshalb sollen auch Name und Sitz der Verlage festgehalten werden."

Die einzelnen Objektklassen werden durch Rechtecke dargestellt und es werden die beschreibenden Attribute an die Rechtecke angetragen. Aus dem betrachteten Ausschnitt ergeben sich die Objekte Buch und Verlag.



Zwischen diesen beiden Objektklassen besteht eine Beziehung. Als Bezeichnung für diese Beziehung soll "gibt heraus" gewählt werden. Als Typ ist (wie erwähnt) 1 : N zu wählen. Attribute, die die Beziehung eventuell näher beschreiben können, sind im betrachteten Beispiel nicht bekannt.



Werden jetzt schrittweise alle erkannten Objekte und Beziehungen in das Diagramm eingezeichnet, so ergibt sich die folgende Darstellung, in welcher allerdings auf das Einzeichnen der Attribute aus Gründen der Übersichtlichkeit verzichtet wurde.

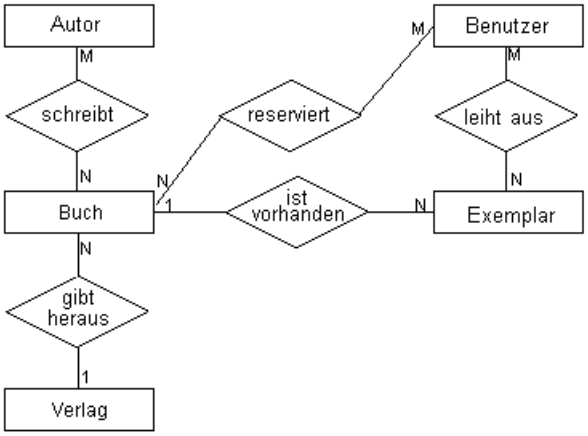
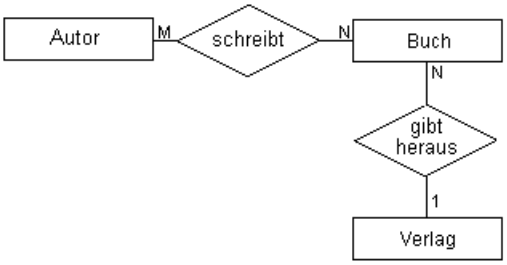


Abbildung 27: Entity-Relationship-Diagramm des Beispiels

Ableitung des relationalen Schemas aus dem Entity-Relationship-Diagramm

Zur Ableitung des relationalen Schemas aus dem ERD werden nun die in oben beschriebenen Regeln verwendet. Als Beispiel soll der folgende Ausschnitt aus dem ERD betrachtet werden:



Für jede der dargestellten Objektklassen ist eine Relation zu erstellen. Dabei sind alle in der Spezifikation angegebenen Attribute in die Relation aufzunehmen und es muss der Primärschlüssel festgelegt werden. Ist mit den angegebenen Attributen die Bildung eines solchen eindeutigen Objektschlüssels nicht möglich oder wird der Schlüssel zu umfangreich, so ist es ratsam, ein zusätzliches Attribut als Primärschlüssel einzuführen. Ein solcher Schlüssel ist zum Beispiel die Autorennummer. Primärschlüssel werden in einem solchen relationalen Schema durch Unterstreichen gekennzeichnet. Es ergeben sich die folgenden Relationen.

```
autor ( autornr, name, vorname, biographie )
buch ( isbn, titel, sprache, auflage, jahr, seitenzahl, format )
verlag ( verlagid, name, sitz )
```

Jetzt müssen noch die Beziehungen modelliert werden. Die zwischen Verlag und Buch bestehende 1 : N-Beziehung wird dadurch abgebildet, dass der Primärschlüssel

der Relation "verlag" als Fremdschlüssel in die Relation "buch" aufgenommen wird. Fremdschlüssel sollen hier kursiv dargestellt werden. Damit hat die Relation "buch" folgendes Aussehen:

buch (isbn, titel, sprache, auflage, jahr, seitenzahl, format, *verlagid*)

Die zwischen den Objekten der Objektklasse Buch und den Objekten der Objektklasse Autor bestehende M : N-Beziehung wird durch das Erzeugen einer neuen Relation, welche die Schlüssel von "buch" und "autor" als Fremdschlüssel enthält, modelliert. Der Primärschlüssel dieser Relation wird durch die Kombination beider Fremdschlüssel gebildet.

schreibt (autornr, isbn)

Werden diese Ableitungsregeln auf das gesamte ERD angewendet, so ergibt sich das folgende relationale Schema der Bibliotheksdatenbank:

Relationen zur Modellierung der Objektklassen

autor (autornr, name, vorname, biographie)

buch (isbn, titel, sprache, auflage, jahr, seitenzahl, format, *verlagid*)

verlag (verlagid, name, sitz)

benutzer (nutzernr, name, vorname, beruf, plz, ort, strasse, telefon, rechte, passwort, sperrkz)

exemplar (*isbn*, lfdnr, praesent, ort)

Relationen zur Modellierung der M : N- Beziehungen

schreibt (*autornr, isbn*)

reserviert (*nutzernr, isbn*)

leiht_aus (*nutzernr, isbn, lfdnr*, leihdat, sollrueck, rueckdat, mahndat)

Um mögliche Fehler beim Erstellen des ERD oder beim Ableiten des relationalen Schemas zu erkennen, sollte das relationale Schema nun noch auf die Einhaltung der Normalformen für das relationale Datenmodell überprüft werden.

4.2.3 Normalformen für relationale Datenbanken

Fehler bzw. Ungenauigkeiten beim Datenbankentwurf können dazu führen, dass ein entwickeltes relationales Schema noch Mehrdeutigkeiten und Inkonsistenzen beinhaltet. Werden diese vor der Implementierung nicht beseitigt, können Redundanzen und Anomalien auftreten, die im Laufe der Zeit zu inkonsistenten Datenbankzuständen führen können.

E. F. Codd [Cod71] formulierte im Rahmen der Theorie der relationalen Datenbanken auch Leitlinien zum Entwurf von Datenbank-Relationen (Normalformen).

Normalisierung

Die Normalisierung beschreibt gewisse Regeln, die ein relationales Schema erfüllen muss. Der wichtigste Punkt ist die Vermeidung von Redundanzen. Durch die Verteilung von Informationen über viele Tabellen wirken relationale Schemata schnell unübersichtlich. Diese Verteilung ist aber notwendig, um die Konsistenz der Daten zu gewährleisten.

Im Rahmen der Normalisierung sollen die Relationen eines Schemas auf die funktionalen Abhängigkeiten ihrer Attribute von den jeweiligen Schlüsselattributen hin untersucht werden. Nicht korrekte Relationen müssen erkannt und i.d.R. in mehrere Relationen aufgeteilt werden (Dekomposition).

Ein relationales Schema wird in der Praxis als weitgehend korrekt angesehen, wenn es den ersten drei Normalformen genügt. Diese Normalformen bauen aufeinander auf. Die weiteren Normalformen sind in der Praxis von untergeordneter Bedeutung. Grundsätzlich sollte ein Entwickler beim Entwurf von relationalen Schemata auch etwas „gesunden Menschenverstand“ gebrauchen und die Normalisierungsregeln nicht als absolut hinnehmen.

1. Normalform

Eine Relation genügt der ersten Normalform, wenn die Werte der Attribute atomar sind. Zusammengesetzte oder kollektionswertige Attribute sind nicht gestattet. Die 1NF ist eine Grundforderung für relationale Datenbanken.

Was als atomar angesehen wird, ist vom konkreten Modell abhängig. Eine aus Postleitzahl, Ort und Straße bestehende Anschrift kann auf drei Attribute verteilt werden. Damit sind z.B. Auswertungen möglich, die sich auf bestimmte Postleitzahlenbereiche beziehen. Die Anschrift kann aber auch einfach in Form einer Zeichenkette gespeichert werden. Dann sind aber z.B. Suchen, die sich auf bestimmte Postleitzahlenbereiche beziehen, mit den Standard-Suchmechanismen nicht mehr möglich.

Beispiel: Zu einem Kunden werden die Kundennummer (`KDNR`), der Name und die Anschrift gespeichert. Die Anschrift besteht aus einer Postleitzahl (`PLZ`), dem Ort und der Straße.

KNDNR	NAME	ANSCHRIFT		
		PLZ	ORT	STRASSE

Varianten der Relation in der 1.Normalform:

KNDNR	NAME	PLZ	ORT	STRASSE

KNDNR	NAME	ANSCHRIFT

Zusammengesetzte oder kollektionswertige Attribute müssen, sofern das DBMS-Produkt keine passenden Erweiterungen (z.B. Listen, Arrays) anbietet, in eigene Relationen ausgelagert werden. Die folgende Abbildung zeigt eine Tabelle mit einem kollektionswertigen Attribut (E-Mail).

MATRIKELNR	NAME	VORNAME	E-MAIL	…
G150123WI	Meier	Franz	franz.meier@dhge.de franz.meier@gmail.com franz@meier.de	
G170235IK	Müller	Klaus	klaus.mueller@dhge.de klausiM@gmx.de	
G180456PI	Schmidt	Claudia	claudia.schmidt@dhge.de	
G160789WI	Schulze	Robert	robert.schulze@dhge.de rschulze@web.de robertschulze@gmail.com schulle@fun.de	

Nach der Normalisierung würden die Daten in zwei Tabellengespeichert, wie unten gezeigt.

MATRIKELNR	NAME	VORNAME	…
G150123WI	Meier	Franz	
G170235IK	Müller	Klaus	
G180456PI	Schmidt	Claudia	
G160789WI	Schulze	Robert	

MAIL_ID	*MATRIKELNR*	E-MAIL
1	G150123WI	franz.meier@dhge.de
2	G150123WI	franz.meier@gmail.com
3	G150123WI	franz@meier.de
4	G170235IK	klaus.mueller@dhge.de
5	G170235IK	klausiM@gmx.de
6	G180456PI	claudia.schmidt@dhge.de
7	G160789WI	robert.schulze@dhge.de
8	G160789WI	rschulze@web.de
9	G160789WI	robertschulze@gmail.com
10	G160789WI	schulle@fun.de

Funktionale Abhängigkeit

Eine funktionale Abhängigkeit zwischen Attributmengen besteht, wenn aus einem Attributwert der Menge A der jeweils korrespondierende Attributwert der Menge B abgeleitet werden kann (Schreibweise A → B – So kann z.B. von der Matrikelnummer eines Studenten auf seinen Namen geschlossen werden. Die Umkehrung muss allerdings nicht gelten). Funktionale Abhängigkeit ist eine Eigenschaft der Semantik der Attribute einer Relation.

Volle funktionale Abhängigkeit

Eine volle funktionale Abhängigkeit eines Attributs von einem Schlüssel liegt vor, wenn es sich um einen aus mehreren Attributen zusammengesetzten Schlüssel handelt und das Attribut von einer aus allen Teilschüsseln bestehenden Wertekombination abhängig ist.

2. Normalform

Eine Relation genügt der 2NF, wenn sie in der 1.NF vorliegt und alle nicht zu den Schlüsselattributen der Relation gehörenden Attribute nur vom gesamten Primärschlüssel der Relation abhängig sind (volle funktionale Abhängigkeit).

Demzufolge sind Relationen, die keinen zusammengesetzten Primärschlüssel besitzen, immer in 2NF.

Beispiel: Eine Tabelle, die Kunden- und Auftragsdaten beinhaltet, enthält die folgenden Attribute:

KNDNR	NAME	PLZ	ORT	STRASSE	WARNR	BEZ	PREIS	ANZ

Den Primärschlüssel bildet die Kombination aus KNDNR und WARNR.
Innerhalb der Relation existieren die folgenden Abhängigkeiten:

- NAME, PLZ, ORT STRASSE sind funktional abhängig von KNDNR
- BEZ und PREIS sind funktional abhängig von WARNR
- ANZ ist von der Kombination aus KDNR und WARNR abhängig (volle funktionale Abhängigkeit

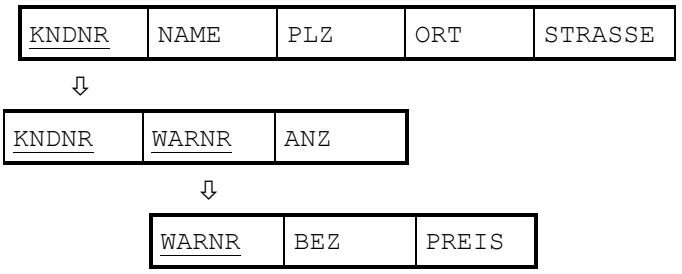
Durch die Verletzung der 2. Normalform können die folgenden Anomalien auftreten.

Einfügeanomalie: ⇨ Informationen über einen zukünftigen Artikel oder Kunden können erst bei Erteilung eines Auftrags eingetragen werden. Weiterhin besteht die Möglichkeit, zu einer Kundennummer unterschiedliche Kundendaten bzw. zu einer Warennummer verschiedene Artikeldaten einzutragen.

Löschanomalie: ⇨ Beim Löschen der Daten eines alten Auftrags werden auch die Informationen über den Kunden und die entsprechenden Waren mit gelöscht.

Änderungsanomalie: ⇨ Durch Redundanzen werden mehrere Änderungsaktionen bezüglich eines Sachverhaltes notwendig (z.B. Änderung des Preises einer bestimmten Ware). Weiterhin besteht die Möglichkeit, zu einer Kundennummer unterschiedliche Kundendaten bzw. zu einer Warennummer verschiedene Artikeldaten einzutragen.

Die 2. NF kann durch die folgende Aufteilung erreicht werden:



Alternativ könnte auch ein neuer, nicht zusammengesetzter („künstlicher“) Schlüssel in die Relation (z.B. eine eindeutige Auftragsnummer) eingefügt wird. Dies würde zwar formal zum Erreichen der 2. Normalform führen, das Problem der Anomalien würde dadurch aber nicht beseitigt. Das Hinzufügen des neuen Schlüssels würde zu einer Verletzung der 3. Normalform führen.

3. Normalform

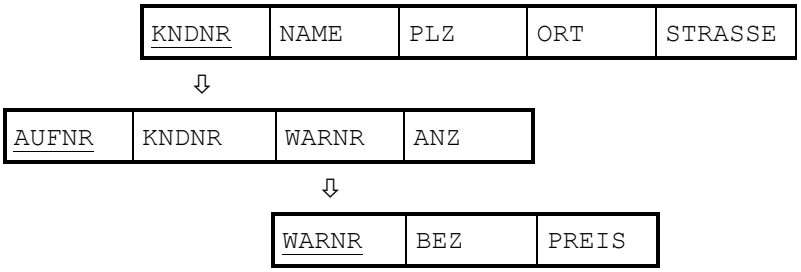
Eine Relation liegt in der 3. NF vor, wenn sie die 2. NF erfüllt und kein Nicht-Schlüsselattribut von einem anderen Nicht-Schlüsselattribut funktional abhängig ist.

Beispiel: Im Beispiel der Auftragstabelle würde das Hinzufügen der Auftragsnummer zu folgender Situation führen.

AUFNR	KNDNR	NAME	PLZ	ORT	STRASSE	WARNR	BEZ	PREIS	ANZ

Lediglich die Attribute KDNR, WARNR und ANZ sind direkt funktional von AUFNR abhängig. Die übrigen Attribute sind nur indirekt (*transitiv*) über die Attribute KDNR bzw. WARNR von der Auftragsnummer abhängig.

Die 3. NF wird durch die folgende Aufteilung erreicht:



Durch die Normalisierung entsteht eine sehr große Zahl an Tabellen. Um logisch in Beziehung zueinander stehende Daten wieder zusammenzuführen, ist eine oft große Anzahl an Join-Operationen nötig, die u.U. hohe Verarbeitungskosten verursachen. Daher werden beim Entwurf von relationalen Schemata tw. gezielt Normalformen ignoriert. Diese Technik der *gezielten Denormalisierung* wird z.B. im Data-Warehouse-Bereich zur Einsparung von Verarbeitungskosten angewendet. Da dort i.d.R. auch nur lesend auf die Daten zugegriffen wird, sollten die oben beschriebenen Anomalien auch bei teilweiser Verletzung der Normalformen kein größeres Problem darstellen.

Die folgenden Design-Regeln sollten beim Entwurf von relationalen Datenbankschemata berücksichtigt werden.

- Jede Tabelle (Relation) sollte intuitiv einem Objekt- oder Beziehungstyp entsprechen. Die Namen der Tabellen sollten aussagekräftig sein. Wenn die Semantik der Relationen und der darin jeweils enthaltenen Attribute klar ist, ist es wesentlich einfacher, das Schema zu verstehen.
- Die Basistabellen eines relationalen Schemas sollten so entworfen werden, dass keine Anomalien auftreten können (Einhaltung der 3. NF).
- Die Zahl der Attribute, die NULL-Werte enthalten dürfen, sollte möglichst gering gehalten werden.
- Die Basistabellen eines relationalen Schemas sollten so entworfen werden, dass die Equi-Joins (nat. Joins) über Kombinationen aus Primär- und Sekundärschlüsseln erfolgen.

4.2.4 Erweiterungen/Varianten in der ER-Modellierung

Im Laufe der Zeit wurden die Darstellungsmittel in der ER-Modellierung erweitert und es haben sich verschiedene Variationen der ER-Darstellungen etabliert. Hier soll ein kurzer Überblick gegeben werden.

Angabe von Kardinalitäten – (min,max)-Notation

In der bisher betrachteten Notation von Entity-Relationschip-Diagrammen existiert keine Unterscheidung zwischen Muss- und Kann-Beziehungen. Deshalb wurden verschiedene Ergänzungen vorgenommen. Eine einfache Erweiterung ist die Angabe

der Kardinalitäten in Form von Intervallen. Dabei wird in jedem Intervall die Mindestanzahl von Objekten des gegenüberliegenden Typs angegeben, mit denen ein Objekt des betrachteten Typs in Beziehung stehen kann und die maximale Anzahl. Ist die maximale Anzahl nicht genau angebbar, so wird hierfür ein Stern verwendet. Bei den Mindestanzahlen steht eine `0` für eine *optionale* und eine `1` für eine *obligatorische* Zuordnung. Bei den Maximalzahlen steht eine `1` für eine *eindeutige* Zuordnung und ein `*` für eine *multiple* Zuordnung. *Abbildung 28* zeigt ein Beispiel.

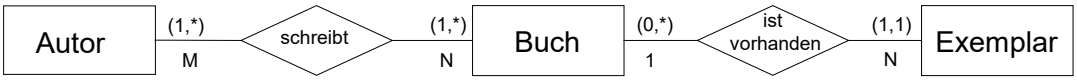


Abbildung 28: ER-Darstellung mit Angabe der Kardinalitäten als Intervall

Bei diesen Intervallangaben ist zu beachten, dass sie bei gängigen Notationen genau entgegengesetzt zur bisher behandelten Notation angegeben werden. Zur Verdeutlichung wurde die Angabe des Beziehungstyps in der bisher behandelten Form unter den Kanten angegeben, die die Beziehung darstellen.

Information-Engineering-Notation (Krähenfußnotation),Pfeil- und MC-Notation

Häufig werden Beziehungen in Varianten von ER-Notationen lediglich als durchgehende Kanten dargestellt. Teilweise wird auch auf die Benennung der Beziehungen verzichtet.

In der MC-Notation steht die Angabe von `m` für *multiple* (mehrfach) und `c` für *choice* (wahlfrei).

In der Information-Engineering-Notation (auch Krähenfußnotation genannt) steht ein `kleiner Kreis` für eine *optionale* und eine `kurze Linie` *obligatorische* Zuordnung. Eine `einfache Kante` steht für eine *eindeutige* und eine sich `verzweigende Kante` steht für eine *multiple* Zuordnung.

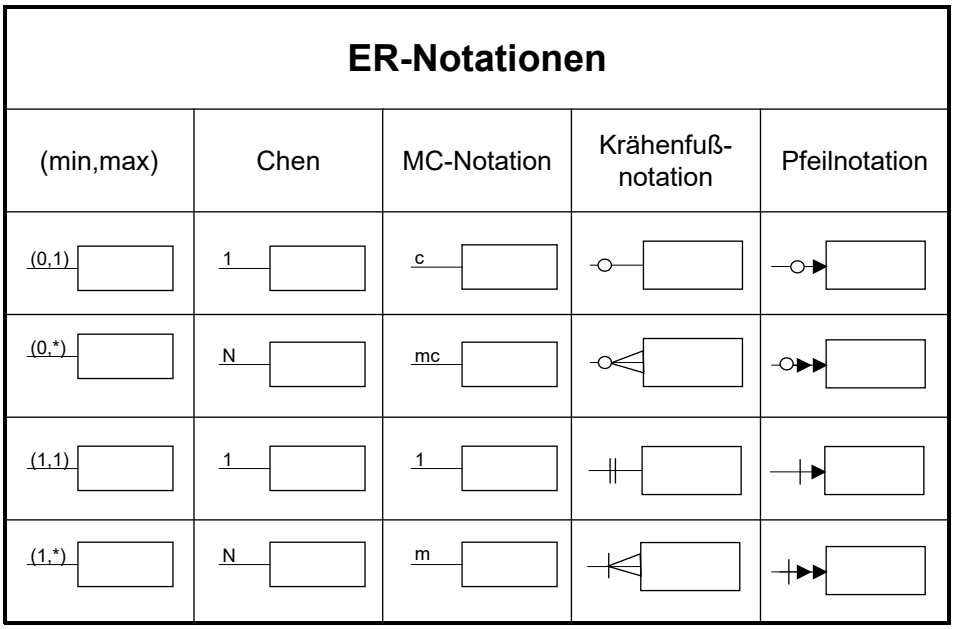


Abbildung 29: Gegenüberstellung verschiedener Notationen

In der Pfeilnotation wird eine *eindeutige Zuordnung* durch eine `einfache`, eine *multiple Zuordnung* durch eine `doppelte Pfeilspitze` dargestellt. *Abbildung 30* zeigt das Beispiel aus Abbildung 29 in der Krähenfußnotation.

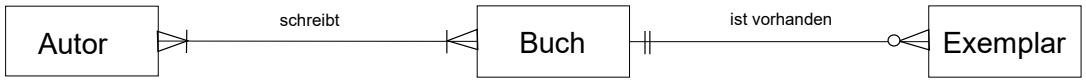


Abbildung 30: ER-Darstellung in Krähenfuß-Notation

Darstellung von Existenzabhängigkeiten – schwache Entity-Typen

Die Existenz bestimmter Objekte kann von der Existenz anderer Objekte abhängen. So existieren bspw. Rechnungsposten nur im Zusammenhang mit der Rechnung, auf der sie vermerkt sind. *Existenzabhängigkeiten* werden bei Beziehungen durch eine doppelte Einrahmung des Bezeichners für die Beziehung gekennzeichnet. Ausgehend vom Bezeichner wird die Kante zum abhängigen Objekttyp ebenfalls als doppelte Linie dargestellt.

Wenn die Objekte eines Typs nicht eindeutig durch ihre eigenen Attribute identifiziert werden können, sondern durch ihre Beziehungen zu anderen Objekten identifiziert werden müssen, wird der entsprechende abhängige Objekttyp als *schwacher Entity-Typ* bezeichnen. Alle anderen heißen *starke Entity-Typen*.

Schwache Entity-Typen werden doppelt eingerahmt dargestellt. Deren Objekte stehen immer in einer Existenzabhängigkeit zu einem identifizierenden Objekt, d.h. zu jedem schwachen Entity muss es ein identifizierendes Entity in der Datenbank geben. Da die Attribute des schwachen Entity allein zur eindeutigen Identifizierung nicht ausreichen, spricht man von einem partiellen Schlüssel, dieser wird gepunktet unterstrichen. Nur zusammen mit dem Schlüssel des identifizierenden Entities ist er ein voller Primärschlüssel. Ein Beispiel zeigt *Abbildung 31*.

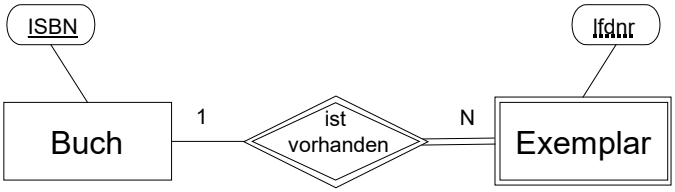


Abbildung 31: ER-Darstellung mit schwachem Entity-Typ

Darstellung von Existenzabhängigkeiten – Globale Normalisierung

Die bei der Entity-Relationship-Modellierung möglichen M:N-Beziehungen müssen bei der Überführung ins relationale Datenmodell aufgelöst werden. Dazu wird ein Beziehungsobjekt eingeführt, dass mit den an der früheren M:N-Beziehung beteiligten Objekten jeweils über 1:N-Beziehungen verknüpft wird (*Abbildung 32*).

Bei der Vorgehensweise, die teilweise auch als *globale Normalisierung* bezeichnet wird, werden bereits auf der Ebene der ER-Modellierung alle M:N-Beziehungen aufgelöst. Damit kann das so entwickelte Datenmodell relativ einfach in ein

relationales Schema überführt werden. Durch eine weitere Variation der Notation der Modellierung wird es weiterhin möglich, auch Existenzabhängigkeiten zwischen den Objekttypen darzustellen. Das Modell kennt unabhängige Objekttypen und von diesen Objekttypen abhängige Objekttypen.

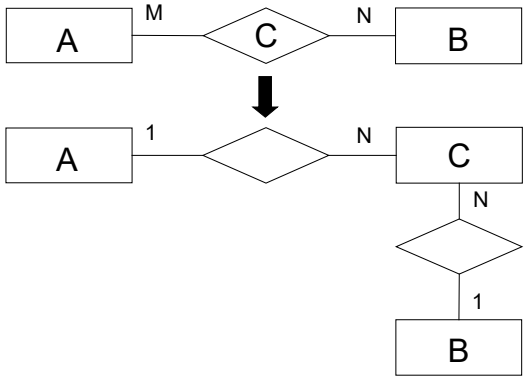


Abbildung 32: Abbildung von M:N-Beziehungen über 1:N-Beziehungen

Die unabhängigen Objekttypen werden links und die abhängigen Objekttypen jeweils rechts von den Objekttypen notiert, von denen sie abhängig sind. Dies ist auch mehrstufig möglich. Die Modellierung erfolgt von links nach rechts. Beziehungen werden über Kanten dargestellt. Es existieren drei verschiedene Beziehungstypen, die in *Abbildung 33* zusammen mit der Notation der jeweiligen Kardinalitäten in der klassischen ER-Notation angegeben sind.

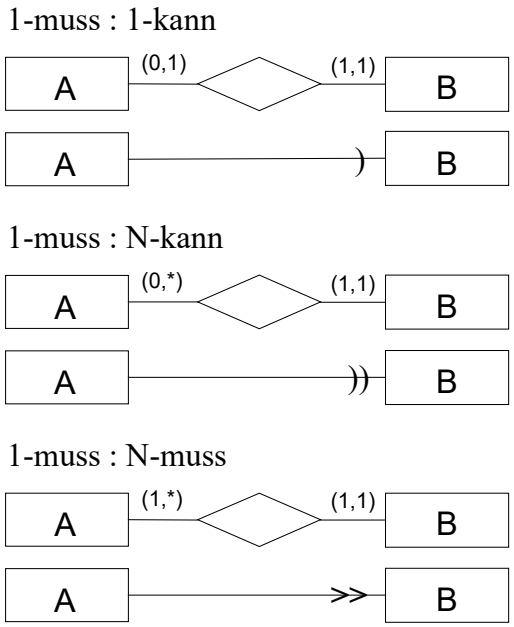


Abbildung 33: Darstellungsmittel bei der globalen Normalisierung

Die Semantik der unterschiedlichen Beziehungstypen lautet wie folgt:

- 1-muss:1-kann – Zu einem unabhängigen Objekt kann maximal ein abhängiges Objekt existieren.

- 1-muss:N-kann – Zu einem unabhängigen Objekt können kein, ein oder mehrere abhängige Objekte existieren.
- 1-muss:N-muss – Zu einem unabhängigen Objekt muss mindestens ein abhängiges Objekt existieren.
- Für alle drei Beziehungstypen gilt, dass zu einem abhängigen Objekt genau ein korrespondierendes unabhängiges Objekt existiert.

Eine 1-muss:1-muss-Beziehung ist nicht vorgesehen, da Objekttypen, zwischen denen eine solche Beziehung bestehen würde, zu einem Objekttyp zusammengefasst werden können.

Abbildung 34 zeigt ein Beispiel eines mittels globaler Normalisierung erstellten Modells. Das Beispiel modelliert einen Ausschnitt der Daten, die zur Abwicklung von Bestellungen nötig sind. Es existieren zunächst die beiden unabhängigen Objekttypen `Kunde` und `Artikel`. Die Kunden können jeweils `Bestellungen` auslösen. Eine Bestellung enthält mindestens einen `Bestellposten`, über den der Bezug zum jeweils bestellten Artikel hergestellt wird.

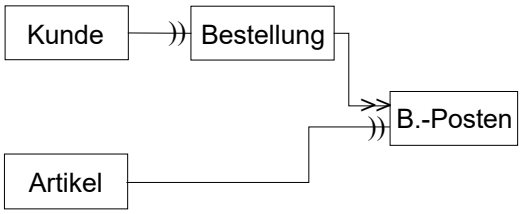


Abbildung 34: Beispiel eines mittels globaler Normalisierung erstellten Modells

Klassische ER-Darstellungen können in mehreren Schritten in die Darstellungen gem. der globalen Normalisierung überführt werden. Diese Vorgehensweise wird tw. auch als *iteratives Auflösen* bezeichnet und umfasst die folgenden Schritte.

- Ermittlung von unabhängigen und abhängigen Objekttypen im ERD und entsprechende Darstellung im normalisierten Modell.
- Eintragen schon normalisierter Beziehungen (1:N) in das normalisierte Modell.
- Auflösen von im ERD nicht normalisierten Beziehungen (M:N) durch Einführung von Beziehungsobjekttypen und Einzeichnen der entsprechenden Beziehungen.
- semantische Analyse des normalisierten Datenmodells
- Bereinigung von Redundanten Beziehungen.

5 Sicherung von Konsistenz und Integrität

<FWB> Konsistenz : Dichtigkeit, Zusammenhang, Beständigkeit

<FWB> Integrität : Makellosigkeit, Unbestechlichkeit

5.1 Konsistenz einer Datenbank

Da eine Datenbank ein Ergebnis einer Modellbildung eines Umweltausschnitts ist, müssen das Modell, seine Daten und die Operationen widerspruchsfrei sein und mit dem abgebildeten Umweltausschnitt im Einklang stehen. Es muss weitestgehend sichergestellt sein, dass das Modell keine undefinierten Zustände (Konsistenzverletzungen) annehmen kann.

Konsistenzverletzungen können innere und äußere Ursachen haben. Innere Ursachen können operationale Fehler sein, zum Beispiel Fehler in den Anwendungsprogrammen oder der Betriebssoftware (Betriebssystem, DBMS, etc.).

Zu den äußeren Ursachen werden bspw. fehlerhafte Eingaben oder Bedienungsfehler gezählt.

Beispiel: Mitarbeiter Schulze hat die `Mitarbeiternummer=0815`. Die Datenerfasserin gibt aber in der Tabelle `GEHALT` die `Mitarbeiternummer=0816` ein. Herr Meier, der diese Nummer hat, kann sich bei der nächsten Gehaltszahlung freuen.

5.2 Integritätsbedingungen

Um Konsistenzverletzungen zu vermeiden, müssen bestimmte Voraussetzungen geschaffen werden, die es dem DBMS ermöglichen, bestimmte Fehler zu erkennen und darauf zu reagieren. Dies wird mit Integritätsbedingungen erreicht, die i.d.R. ein Teil des jeweiligen Datenbankschemas sind.

Beispiele für Integritätsbedingungen:

- Integritätsbedingung bezüglich eines Attributs –z.B. Gehalt ist größer als 0 (`CHECK`)
- Integritätsbedingung bezüglich mehrerer Tupel – z.B. Gehalt der Mitarbeiter muss kleiner als das des Direktors sein (Trigger)
- Integritätsbedingung bezüglich einer Tupelmenge – z.B. das Durchschnittsgehalt aller Mitarbeiter muss immer 1800,- € sein (Trigger)
- Integritätsbedingung bezüglich der Beziehungen zwischen Relationen – Die Menge der Abteilungsleiter muss immer eine Teilmenge der Menge Mitarbeiter sein (`FOREIGN KEY`).

Integritätsbedingungen können sowohl einen konsistenten Zustand, als auch Übergangsbedingungen beschreiben – z.B. es gibt nur Gehaltserhöhungen.

Bei der Nutzung von Integritätsbedingungen treten drei Fragen auf :

1. Wie werden Integritätsbedingungen formuliert?
2. Wann sollen die Integritätsbedingungen geprüft werden?
3. Wie soll auf eine Integritätsverletzung reagiert werden?

Einige allgemeine Integritätsbedingungen können beim Anlegen von Schemaelementen formuliert werden.

Beispiel : die Angabe von `NOT NULL`

```
CREATE TABLE tabellenname
(    attribut1  INTEGER NOT NULL,
     attribut2   CHAR,
     attribut3   INTEGER
)
```

`NOT NULL` legt fest, dass das entsprechende Attribut zwingend einen Wert erhalten muss.

Aufwendige Integritätsbedingungen müssen u.U. vom Anwendungsprogrammierer selbst realisiert werden. Er ist für die Formulierung der Integritätsbedingungen entsprechend den Konsistenzerfordernissen verantwortlich.

Die Prüfung der Einhaltung der Integritätsbedingungen kann zum Beispiel bei jeder Update-Operation, periodisch (stündlich, täglich...) oder auf Anforderung (z.B. vom DBA) erfolgen.

Als Reaktion auf Verletzungen der Integritätsbedingungen können bspw. Fehlermeldungen erzeugt, die Ausführung von Operationen abgelehnt oder selbstständige Korrekturen durch Programme durchgeführt werden.

5.3 Transaktionen

Zur Begrenzung der negativen Auswirkungen von intern verursachten Fehlern wurden in die Datenbanktechnologie Transaktionskonzepte einbezogen. Transaktionen – im Sinne der Datenbanktechnologie – sind Update-Operationen (im weitesten Sinne) auf der Datenbank. Diese Update-Operationen verändern Werte in der Datenbank (Löschen, Hinzufügen und Ändern von Datensätzen). Eine Transaktion überführt eine Datenbank immer von einem konsistenten Zustand in einen anderen konsistenten Zustand. Innerhalb einer Transaktion kann sich die jeweilige Datenbank in einem inkonsistenten Zustand befinden.

Beispiel: Zu einer Banküberweisung gehören immer die Belastung eines Kontos und die entsprechende Gutschrift auf ein anderes Konto. Dies erfordert im einfachsten Fall die Änderung von zwei verschiedenen Datensätzen. Wenn die Datenbank vor der Überweisung in einem konsistenten Zustand ist, ist sie nach der Änderung des ersten Datensatzes in einem inkonsistenten Zustand, weil die Geldmenge nicht mehr korrekt ist. Erst nachdem beide Datensätze geändert wurden, und die Änderungen permanent

in der Datenbank erfolgt sind, ist die Operation (Transaktion) abgeschlossen und die Datenbank wieder in einem konsistenten Zustand.

Eine Transaktion im Sinne der Datenbank-Technologie muss die ACID-Eigenschaften erfüllen. ACID steht für:

1. Atomarität
2. Konsistenz
3. Isolation
4. Dauerhaftigkeit

Zur Verwaltung der Transaktionen führen Datenbank-Management-Systeme intern Transaktionstabellen, deren Einträge üblicherweise über eine Transaktionsnummer identifiziert werden.

5.3.1 Atomarität

Eine Transaktion ist atomar (unteilbar). Das heißt, auch wenn eine Transaktion in mehreren Teilschritten erfolgen muss, wie im obigen Beispiel, so werden diese Teilschritte als eine ganzheitliche Operation betrachtet. Damit dauert die Transaktion solange, bis die Datenbank von einem konsistenten Zustand in den anderen konsistenten Zustand überführt wurde. Die einzelnen Datenbankanweisungen, aus der sich die Transaktion zusammensetzt, werden üblicherweise nacheinander ausgeführt. Kann ein Teil der Transaktion (eine oder mehrere Operationen) nicht erfolgreich ausgeführt werden, so werden typischerweise alle von der Transaktion bisher vorgenommenen Änderungen wieder zurückgesetzt (ROLLBACK).

Unter dem Aspekt der Konsistenz ist eine *Transaktion* eine logisch abgeschlossene Arbeitseinheit, die eine Datenbank *von einem konsistenten Zustand in einen anderen konsistenten Zustand überführt.*

Für Anwendungsprogrammierer bedeutet dies i.d.R., dass sie *transaktionsorientiert programmieren* müssen. Datenbanksprachen enthalten normalerweise Anweisungen zur Transaktionssteuerung. Hier kann der Programmierer den Umfang einer Transaktion kennzeichnen, für die dann auch die Transaktionseigenschaften gelten. Wie die Kennzeichnung von Transaktionen erfolgen muss, hängt vom (vom DBMS) verwendeten Transaktionskonzept ab.

Die `BEGIN WORK`-Anweisung zur Kennzeichnung des Beginns einer Transaktion kann nur ausgeführt werden, wenn keine Transaktion in der aktuellen Sitzung aktiv ist.

Bei *ANSI-kompatiblen Datenbanken* werden alle Operationen implizit in Transaktionen ausgeführt. Der Beginn einer Transaktion muss nicht gekennzeichnet werden. Allerdings muss die implizit laufende Transaktion nach Abschluss der jeweiligen logisch zusammengehörigen Arbeitseinheit abgeschlossen werden. Bei Datenbanken, die nicht im ANSI-kompatiblen Transaktionsmodus arbeiten, ist jede einzelne SQL-Anweisung für sich als einzelne Transaktion realisiert (Autocommit). Müssen mehrere SQL-Anweisungen zu einer Transaktion zusammengefasst werden, so sind Transaktionsbeginn und Transaktionsende zu kennzeichnen.

Auf eine Datenbank angewendete Änderungsoperationen werden im Transaktionsprotokoll vermerkt. Mit Hilfe dieses Protokolls kann nach einem Transaktionsabbruch ein konsistenter Datenbankzustand wiederhergestellt werden.

Transaktionstypen:

- Autocommit – Jede einzelne Anweisung ist eine Transaktion.
- Implizite Transaktion – Eine neue Transaktion wird implizit gestartet, sobald die vorhergehende Transaktion abgeschlossen ist. Jede Transaktion muss jedoch explizit mit `COMMIT` oder `ROLLBACK` beendet werden.
- Explizite Transaktion – Jede Transaktion wird explizit mit `BEGIN TRANSACTION` gestartet und explizit mit `COMMIT` oder `ROLLBACK` beendet.

5.3.2 Isolation/Abkapselung

Alle Änderungen in der Datenbank, die eine Transaktion bewirkt, sollen erst nach deren Beendigung für andere Transaktionen sichtbar werden. Diese Eigenschaft ist für die Synchronisation von konkurrierenden Zugriffen wichtig, da auf eine Datenbank in der Regel mehrere Nutzer lesend oder schreibend zugreifen. Durch die Eigenschaft der Isolation wird verhindert, dass sich in Ausführung befindliche Transaktionen gegenseitig beeinflussen. Dies wird beispielsweise durch spezielle Sperrprotokolle oder Zeitstempelverfahren realisiert. Ohne die Eigenschaft der Isolation würden bspw. Zwischenzustände und damit auch für andere als die verursachenden Transaktionen, inkonsistente Datenbankzustände sichtbar.

Ein einfaches **Beispiel** für einen möglichen Fehler stellen sog. *Lost Updates* dar. Zwei Prozesse lesen und schreiben fast gleichzeitig auf denselben Datensatz:

Kontostand alt:			**400 €**
	Prozess 1	Prozess 2	
Prozess 1 liest	400 €		
Prozess 2 liest		400 €	
Prozess 1 bucht	+200 €		
Prozess 1 schreibt	600 €		
Prozess 2 bucht		-50 €	
Prozess 2 schreibt		350 €	
Kontostand neu:			**350 €**

W o s i n d d i e 2 0 0 € ?

Folgende Fehlersituationen sind bspw. bei (mindestens zwei) parallel auf denselben Daten arbeitenden Transaktionen denkbar:

- beide greifen nur lesend zu → keine
- einer lesend und einer schreibend → falsche Ergebnisse (dirty read)
- beide schreibend → inkonsistenter Zustand möglich

Die Zugriffe auf eine Datenbank müssen synchronisiert werden. Eine Möglichkeit ist das Sperren von Teilen der Datenbank, auf die eine Transaktion zugreift. Diese DB-Teile werden erst nach Beendigung der Transaktion wieder freigegeben.

Dabei kann zwischen Sperren für lesenden oder schreibenden Zugriff unterschieden werden. Während beim schreibenden Zugriff der gesperrte Teil für *alle* anderen Transaktionen gesperrt sein (exklusive Sperre) muss, brauchen die jeweiligen Daten beim lesenden Zugriff nur für schreibende Transaktionen gesperrt werden.

Wird eine Transaktion durch eine Sperre behindert, so existieren verschiedene Möglichkeiten einer Reaktion. Die Transaktion kann

- als fehlerhaft abgebrochen werden,
- auf das Aufheben der Sperre warten oder
- die Sperre ignorieren.

Das genaue Verhalten der Transaktion wird mit der jeweiligen *Isolationsebene* festgelegt.

Eine Menge konkurrierender Transaktionen gilt als synchronisiert, wenn deren parallele Abarbeitung die gleichen Ergebnisse liefern, wie deren sequenzielle Bearbeitung.

5.3.3 Dauerhaftigkeit

Das Ergebnis einer Transaktion ist dauerhaft. Die Ergebnisse einer erfolgreich abgeschlossenen Transaktion bleiben dauerhaft in der Datenbank erhalten. Diese Forderung kommt insbesondere nach *Systemfehlern* („Systemabstürzen“) zum Tragen.

Zur Verbesserung der Systemleistung erfolgen Datenbankoperationen über dem Datenbankpuffer. Das heißt, Anwendungsprogramme greifen immer auf Daten im Datenbankpuffer zu. Es ist die Aufgabe der Pufferverwaltung, auf eine möglichst effiziente Weise die benötigten Daten dort zur Verfügung zu stellen. Schreiboperationen werden von DBMS-Produkten i.d.R. asynchron ausgeführt. Ein geänderter Block wird dabei nicht sofort nach der Änderung auf den entsprechenden Datenträger geschrieben, sondern verbleibt zunächst im Puffer und wird lediglich zum Schreiben „vorgemerkt“. Dazu kann er bspw. in eine sog. *Page Cleaner List* eingetragen werden. Das eigentliche Schreiben der geänderten Blöcke wird dann von separaten Prozessen bzw. Threads (sog. *Page Cleaner Agents* [IBM04]) durchgeführt. Solche Schreiboperationen werden z.B. initiiert, wenn die Anzahl

Einträge in der Page Cleaner List einen vorgegebenen Schwellwert übersteigt oder ein bestimmtes Zeitintervall seit der letzten Schreiboperation vergangen ist.

Wenn also eine Transaktion abgeschossen wird, so wurden nicht zwangsläufig auch alle im Datenbankpuffer vorgenommenen Änderungen dauerhaft auf den Datenträgern vorgenommen. Im Falle des Verlusts des Pufferinhalts oder der Page Cleaner List (z.B. durch einem Stromausfall), sind die zunächst nur im Puffer vorgenommenen Änderungen verloren. Die Datenbank ist mit hoher Wahrscheinlichkeit in einem inkonsistenten Zustand. Die anschließend notwendige Überführung der Datenbank in einen konsistenten Zustand durch Fehlerbehebung wird als *Recovery* oder *Wiederherstellung* bezeichnet.

Dabei werden zwei Arten von Recovery-Operationen unterschieden.

- Beim Vorwärts-Recovery (*rollforward*) wird die Datenbank in einen (relativ zum Ausgangspunkt des Recoverys gesehen) neuen konsistenten Zustand überführt.
- Beim Rückwärts-Recovery (rollback) wird die Datenbank in einen alten konsistenten Zustand versetzt.

5.4 Recovery-Verfahren

Grundvoraussetzung für Recovery-Operationen sind Aufzeichnungen über Datenbankinhalte (*physisches Logging*) und die an den Daten vorgenommenen Veränderungen (*logisches Logging*). Diese Informationen befinden sich im Datenbank-Protokoll (Log). *Abbildung 35* zeigt verschiedene Formen der Datenbank-Loggings.

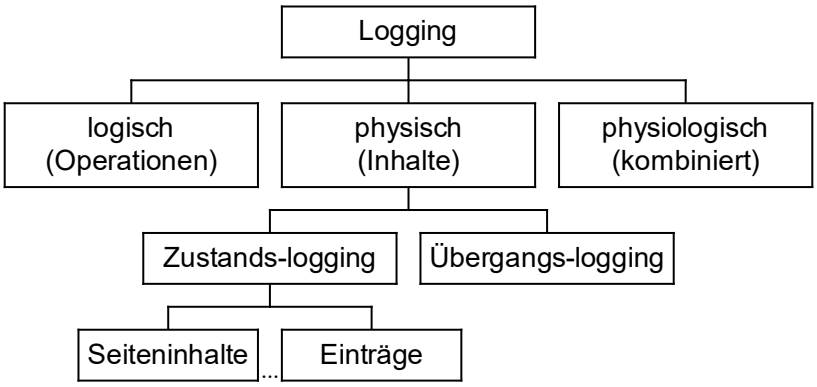


Abbildung 35: Protokollierungsformen

Die Protokollierung kann temporär oder dauerhaft erfolgen. Bei der *temporären Protokollierung* (z.B. in den Rollback-Segmenten bei Oracle) werden einzelne Arbeitsschritte und Datenbankzustände gesichert. Meist geschieht das nur vom Beginn einer Transaktion bis zu deren fehlerfreien Abschluss. Die Protokollierung kann dabei logisch oder physisch erfolgen.

Dauerhafte Protokolle werden über einen längeren Zeitraum hinweg in sog. Log-Dateien aufgezeichnet. Diese werden i.d.R. gesichert (Backup) und zusammen mit einer Gesamtsicherung der Inhalte einer Datenbank aufbewahrt. Aus der

Gesamtsicherung der Datenbank und den gesicherten Log-Dateien kann die Datenbank bspw. auch nach schwerwiegenden Fehlern (z.B. Datenverlust durch Plattenfehler) restauriert werden. Dabei wird unter Nutzung der Datenbanksicherung zunächst ein älterer Stand eingespielt. Anschließend werden die gesicherten Log-Dateien verarbeitet und dabei alle dort aufgezeichneten Änderungen noch einmal ausgeführt (*rollforward*), bis ein möglichst aktueller Stand erreicht ist. Nach dem Beenden der Rollforward-Operation werden noch offene Transaktionen zurückgesetzt. Somit kann unter der Voraussetzung, dass alle Logs bis zum Systemfehler vorhanden sind, der letzte konsistente Stand vor den Systemfehler erreicht werden.

Die Einträge in den Protokollen (*log records*) bestehen meist aus einem Teil mit fester Länge und einem Teil mit variabler Länge. Der feste Teil enthält Informationen, wie bspw. die Nummer des Log-Eintrags (*Log Sequence Number – LSN*), die Nummer der zugehörigen Transaktion, den Typ des Log-Eintrags, die Nummer des Blocks, an dem die Änderung vorgenommen wurde, die Identifikation des betroffenen Datensatzes (bspw. dessen RID) oder Indexeintrags und evtl. die Nummer des vorherigen Log-Eintrags derselben Transaktion. Der variabel lange Teil enthält Informationen über den alten (*Undo Image*) bzw. neuen Inhalt (*Redo Image*) des Teils der Datenbank, auf den sich der Log-Eintrag bezieht (z.B. Datensatz oder Indexeintrag). Ob ein Undo- oder ein Redo-Image protokolliert werden muss, hängt von der Änderungsoperation ab.

- `INSERT` – Redo Image
- `UPDATE` – Undo Image + Redo Image
- `DELETE` – Undo Image

Es werden die folgenden Typen von Einträgen unterschieden:

- Transaktionsbeginn, erfolgreiches Beenden einer Transaktion, Abbruch einer Transaktion und Rücksetzen
- Log-Sätze zur Protokollierung von Änderungen
- Log-Sätze zur Beschreibung von Sicherungspunkten (Checkpoints)

Die Sätze der ersten Gruppe dienen hauptsächlich der Zuordnung der Log-Sätze zu einzelnen Verarbeitungseinheiten. Werden vom DBMS Checkpoints ausgeführt, so werden hauptsächlich die im Datenbankpuffer geänderten Inhalte von Datenblöcken auf die eigentlichen Datenträger geschrieben. Solche Sicherungspunkte werden im Log vermerkt, da sie Aufsetzpunkte für Wiederherstellungsoperationen darstellen. Die Erstellung bzw. Verarbeitung der Sätze, die die Änderungen an den Datenbankinhalten beschreiben, ist in *Abbildung 36* dargestellt.

Normale Änderungen an Datenbankinhalten (`DO`) überführen die Datenbank von einem alten in einen neuen Zustand. Zusätzlich werden zwei Log-Sätze erzeugt. Der Redo-Log-Satz wird für eine Rollforward-Operation (z.B. im Rahmen einer Systemwiederherstellung) genutzt. Der `UNDO`-Satz wird benötigt, um im Falle eines Transaktionsabbruchs die bisher von der Transaktion verursachten Änderungen rückgängig zu machen. Deshalb ist auch eine Zuordnung der Log-Sätze zu den

einzelnen Transaktionen notwendig. Das Rücksetzen selbst wird im Log mit einem *Compensation Log Record* (CLS) vermerkt. Nachdem eine Transaktion beendet wurde, kann bspw. die UNDO-Information zur Reduzierung der Größe des Logs gelöscht werden.

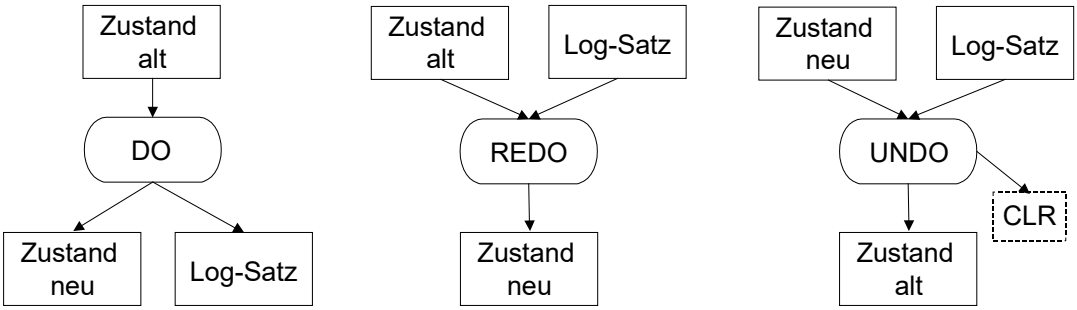


Abbildung 36: Zusammenhang von Datenbank-Operationen und Log-Einträgen

5.5 Sperrkonzepte

5.5.1 Sperrverfahren

Um beim parallelen Zugriff vom mehreren Anwendungen auf die Inhalte einer Datenbank die Eigenschaft der Isolation zu realisieren und Verfälschungen zu vermeiden, müssen Datenbankinhalte gesperrt werden können. Den Sperrverfahren liegen üblicherweise folgende Prinzipien zu Grunde:

- Jedes zu referenzierende Objekt ist vor dem Zugriff zu sperren.
- Die Sperren anderer Transaktionen müssen beachtet werden. Bei Unverträglichkeit der Sperren muss gewartet werden.
- Transaktionen fordern keine Sperren an, die sie bereits besitzen.
- Es ist ein strenges zweiphasiges Sperrprotokoll zu verwenden. Eine Transaktion muss in einer *Wachstumsphase* alle ihre Sperren anfordern. In dieser Zeit darf sie keine Sperren freigeben. Am Ende der Transaktion sind alle Sperren freizugeben (*Schrumpfungsphase*).

Ein sehr einfaches Sperrverfahren ist das RX-Sperrverfahren. Bei diesem Verfahren werden beim Lesen R-Sperren auf den betroffenen Datenbankinhalten errichtet und beim Schreiben X-Sperren (e*X*clusive).

angeforderte Sperre \ vorhandene Sperre	NL	R	X
R	+	+	%
X	+	%	%

Tabelle 1: Kompatibilität beim RX-Sperrverfahren

Die in *Tabelle 1* mit „+“ gekennzeichneten Sperranforderungen können realisiert werden. Bei den mit „%“ gekennzeichneten Anforderungen müssen erst die entsprechenden vorhandenen Sperren aufgehoben werden. „NL“ steht für „No Lock“.

Grundsätzlich besteht durch das Sperren von Ressourcen die Gefahr von *Deadlocks*. Wenn Transaktionen bei Sperranforderungen auf das Aufheben nicht kompatibler Sperren warten, können Deadlocks auftreten. Bei einem Deadlock blockieren verschiedene Transaktionen einander durch die von Ihnen gehaltenen Sperren. Deadlocks können mit Hilfe von sog. *Wartegraphen* erkannt werden (*Abbildung 37*).

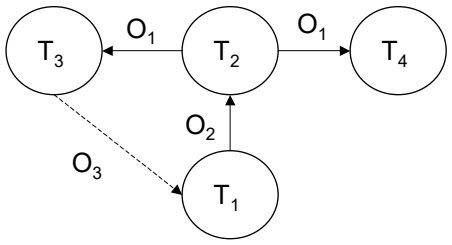


Abbildung 37: Wartegraph für einen Deadlock [HR01]

Im Beispiel in Abbildung 37 versucht die Transaktion T_2 das Objekt O_1 zu sperren. Die Sperranforderung verursacht einen Konflikt mit den Transaktionen T_3 und T_4, die Lesesperren auf O_1 halten. T_2 muss warten. Die Transaktion T_1 versucht eine Sperre auf O_2 zu errichten. Das Objekt ist bereits von T_2 gesperrt. Damit muss T_1 warten. Versucht jetzt die Transaktion T_3 das Objekt O_3 zu sperren, das bereits von T_1 gesperrt wurde, so kommt es zum Deadlock ($T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$).

Zur Vermeidung von Deadlocks sind z.B. die folgenden Vorgehensweisen denkbar.

- Eine Transaktion, die einen Sperrkonflikt verursacht, wird abgebrochen und die Anwendung versucht später erneut, die Transaktion auszuführen.
- Eine Transaktion wartet nur eine bestimmte Zeit auf das Aufheben von Sperren, danach wird sie abgebrochen (Timeout-Verfahren).

Der erreichbare Parallelitätsgrad des Datenbankbetriebs hängt auch von der Größe der gesperrten Bereiche ab (*Sperrgranulate*). Einerseits verursacht jeder Sperre Verwaltungsaufwand, andererseits behindern große gesperrte Bereiche die parallele Verarbeitung. Typischerweise werden Sperren auf Satz-, Block- oder Tabellenebene (samt der betroffenen Indexteile) errichtet. Hierarchische Sperrverfahren versuchen, einen Kompromiss zwischen Verwaltungsaufwand und paralleler Verarbeitung zu finden.

Zum Realisieren von Sperrkonzepten sind zwei Arten von Sperren möglich. Mit physischen Sperren können verschiedene Datenbankbereiche (Sätze, Blöcke, Segmente, etc.) belegt werden. Allerdings können bestimmte Probleme nicht mit *physischen Sperren* gelöst werden. Werden bspw. während einer Transaktion, die sich auf mehrere Datensätze bezieht, die ein bestimmtes Suchprädikat erfüllen, in einer parallel ablaufenden Transaktion neue Datensätze eingefügt, die das Suchprädikat der ersten Transaktion erfüllen, so kann dies inkonsistente Daten der ersten Transaktion zur Folge haben. Das folgende einfache Beispiel soll dies verdeutlichen. Dabei soll eine monatliche Prämie von 5.000 € gleichmäßig auf alle Mitarbeiter einer Abteilung verteilt werden.

Zeit	Transaktion 1	Transaktion 2
1	`SELECT COUNT(*) INTO z FROM mitarbeiter WHERE abteilung='Forschung';`	
2		`INSERT INTO mitarbeiter VALUES('Meier','Karl'...,'Forschung'); COMMIT;`
3	`UPDATE mitarbeiter SET bezuege=bezuege + 5000/z WHERE abteilung='Forschung'; COMMIT;`	

Tabelle 2: Beispiel für das Phantom-Problem

Werden keine Vorkehrungen getroffen, würden mehr als die zur Verfügung stehenden 5.000 € verteilt werden. Der neu hinzugefügte Mitarbeiter wird bei der Zählung der Mitarbeiter der Abteilung noch nicht mit berücksichtigt, aber bei der Verteilung der Prämie (*Phantom-Problem*).

Zur Vermeidung solcher Probleme werden *Prädikatsperren* verwendet. Die allg. Syntax für eine Sperranforderung kann wie folgt aussehen:

`LOCK(<Tabelle>, <Prädikat>, <Sperrmodus>)`

Ein Sperrkonflikt besteht dann, wenn die Prädikate der Sperranforderungen von zwei Transaktionen nicht disjunkt sind.

Wenn im obigen Beispiel ein Index über dem Attribut `abteilung` existiert, so könnte das Phantom-Problem auch dadurch vermieden werden, dass die Indexeinträge mit dem Schlüsselwert `'Forschung'` gesperrt werden.

5.5.2 Isolationsebenen in SQL92

Bei der parallelen Ausführung von Transaktionen können Probleme durch deren gegenseitige Beeinflussung auftreten. Mit den Isolationsebenen, die ursprünglich in [GLP+76] (Stufe 0-3) definiert wurden, können Beeinflussungen durch nebenläufige Transaktionen ausgeschlossen werden. I.d.R. ist hier zu beachten, dass das Ausschließen von Beeinflussungen zu höherem Aufwand und zur Behinderung der parallel laufenden Transaktionen führt.

- `READ UNCOMMITTED` – lässt *Dirty Reads* zu. Das entspricht der Isolationsstufe 0. Das heißt, dass keine gemeinsamen Sperren ausgegeben und keine exklusiven Sperren berücksichtigt werden. Wenn diese Option festgelegt ist, können Daten, für die (noch) kein Commit ausgeführt wurde oder die nicht mit den gültigen Werten übereinstimmen, gelesen werden, Datenwerte geändert werden und Tupel einer Treffermenge zum Transaktionsende nicht mehr existieren oder mittlerweile Tupel hinzugefügt worden sein, die zur Treffermenge gehören. Von den vier Isolationsstufen ist diese die am wenigsten restriktive.
- `READ COMMITTED` – gibt an, dass gemeinsame Sperren aufrechterhalten werden, während die Daten gelesen werden (Isolationsstufe 1). Damit lassen sich *Dirty Reads* vermeiden. Daten können vor dem Ende der aktuellen Transaktion geändert werden. Dies kann zu den Problemen *Non Repeatable Read* oder *Phantom Read* führen.

- REPEATABLE READ – für alle Daten, die in einer Abfrage verwendet werden, werden Sperren errichtet. Diese verhindern, dass andere Benutzer die Daten aktualisieren (Isolationsstufe 2). Jedoch können neue *Phantom-Tupel* von anderen Benutzern in eingefügt und in späteren Lesevorgängen in die aktuelle Transaktion eingeschlossen werden. Diese Isolationsstufe kann zu stärkeren Behinderungen der parallelen Ausführung von Transaktionen führen.
- SERIALIZABLE – errichtet eine Bereichssperre auf der Treffermenge. Diese verhindert, dass andere Benutzer Tupel der Treffermenge aktualisieren oder Tupel in die Treffermenge einfügen, bevor die Transaktion abgeschlossen ist. Von den 4 Isolationsstufen ist diese die restriktivste und kann zu relativ starken Beeinträchtigungen beim parallelen Ausführen von Transaktionen führen.

5.6 2-Phasen-Commit-Protokoll

Wenn Transaktionen auf mehreren Verarbeitungsknoten verteilt ausgeführt werden müssen, müssen zur Sicherstellung der Atomarität mehrere Prozesse koordiniert werden. Eine übliche Verfahrensweise stellt das 2-Phasen-Commit-Protokoll dar. Die wesentlichen Schritte laufen dabei am Ende einer Transaktion ab.

1. Der Koordinator (das ist üblicherweise der Prozess, der das Transaktionsende einleitet) sendet eine PREPARE-Anforderung an alle beteiligten Prozesse (Agenten), um sie zur Beendigung ihrer Teiltransaktionen aufzufordern und deren COMMIT-Ergebnis zu erfragen.
2. Jeder beteiligte Agent sichert seine bisher noch ungesicherten Log-Daten und schreibt einen PREPARED-Satz in das Log. Anschließend übermittelt er dem Koordinator die Nachricht READY oder FAILED. Im Falle von FAILED wird die lokale Transaktion zurückgesetzt.
3. Empfängt der Koordinator nur READY-Nachrichten, so schreibt er einen COMMIT-Satz in sein Log und sendet den Agenten eine COMMIT-Nachricht. Empfängt der Koordinator mindestens eine FAILED-Nachricht, so schreibt er einen ROLLBACK-Satz in sein Log und sendet allen Agenten eine ROLLBACK-Nachricht.
4. Empfangen die Agenten eine COMMIT-Nachricht, so schreiben sie jeweils einen COMMIT-Satz in ihr Log. Empfangen die Agenten eine ROLLBACK-Nachricht, so setzen sie ihre lokalen Transaktionen zurück und schreiben jeweils einen ROLLBACK-Satz in ihr Log. Abschließend senden die Agenten eine ACK-Nachricht an den Koordinator, der nach dem Empfang aller ACK-Nachrichten die Transaktion beendet.

5.7 Optimistische Synchronisation

Neben Sperrverfahren sind auch optimistische Synchronisationsverfahren denkbar. Solche Verfahren gehen von der Annahme aus, dass *Konflikte selten* auftreten. Es werden keine Sperrungen vorgenommen und erst am Ende einer Transaktion wird

geprüft, ob Konflikte aufgetreten sind. Eine Transaktion läuft dabei in drei wesentlichen Phasen ab.

1. In der sog. *Lesephase* läuft die eigentliche Transaktionsverarbeitung im Puffer ab.
2. Die *Validierungsphase* wird mit der Aufforderung zum Beenden der Transaktion gestartet. Hier wird geprüft, ob die Transaktion im Konflikt mit anderen Transaktionen steht. Ist das der Fall, so wird die aktuelle Transaktion zurückgesetzt.
3. Sind keine Konflikte aufgetreten, so werden in der *Schreibphase* die eigentlichen Änderungen durchgeführt (Erzeugen der Log-Einträge, Einbringen der Änderungen in die Datenbank).

6 Datenspeicherung und Datenzugriffe

6.1 Speicherhierarchie

Zwischen den verschiedenen Baugruppen von Computersystemen bestehen sehr große Unterschiede bezüglich ihrer Leistungsfähigkeit. Leider gilt dies auch für die Kosten. Die leistungsfähigsten Komponenten sind meist wesentlich teurer als die weniger leistungsfähigen Teile. Daher sind i.d.R. auch die Kapazitäten der schnelleren Speicherelemente deutlich geringer als die der langsameren Bausteine. Wie aus *Abbildung 38* ersichtlich ist, unterscheiden sich die Zugriffsgeschwindigkeiten von Externspeichermedien, wie z.B. Festplatten und dem Arbeits- oder Hauptspeicher besonders stark. Dieser Unterschied in der Zugriffsgeschwindigkeit wird auch als *Zugriffslücke* bezeichnet. Diese ist so groß, dass die Vernachlässigung der Kosten „oberhalb“ der Zugriffslücke bei der Abschätzung des zur Abarbeitung von Datenbankoperationen notwendigen Aufwands eine durchaus gängige Praxis darstellt.

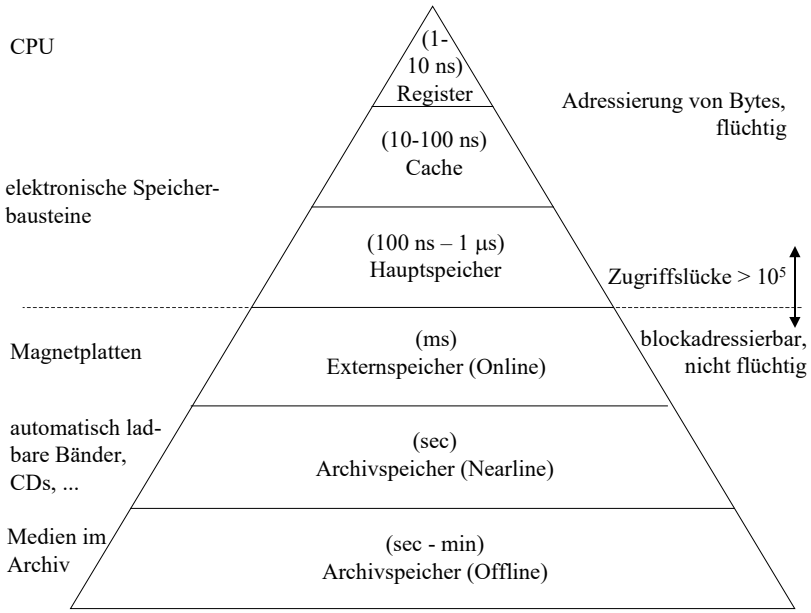


Abbildung 38: Speicherhierarchie nach [GR93]

Um Zugriffe bei Datenbank-Management-Systemen zu beschleunigen, ist es wichtig, die Auswirkungen der Zugriffslücke zu verringern. Dies heißt, die Zahl der Zugriffe auf Externspeichermedien möglichst gering zu halten. Bei Datenbank-Management-Systemen haben die verwendeten Speicherungs- und Zugriffsstrukturen sowie die Mechanismen zur Verwaltung des Datenbankpuffers starken Einfluss auf die Anzahl der zur Abarbeitung einer Datenbankoperation notwendigen Externspeicherzugriffe.

Um die Auswirkungen der unterschiedlichen Arbeitsgeschwindigkeiten der verschiedenen Ebenen der Speicherhierarchie weiter zu verringern, wird üblicherweise ein mehrstufiges Pufferungskonzept angewendet. Wie aus *Abbildung 39* zu erkennen ist, müssen zu verarbeitende Daten mehrere Ebenen durchlaufen. Dies ist besonders beim permanenten Einbringen von Änderungen an Datenbankinhalten zu berücksichtigen.

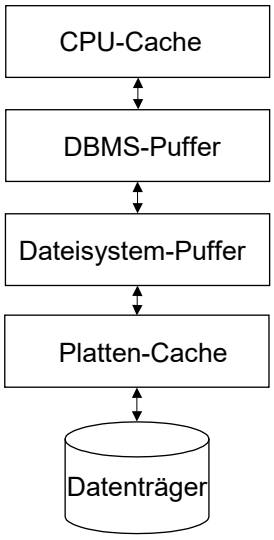


Abbildung 39: Pufferungsebenen

6.2 Datenträger- und Pufferverwaltung

Die Verwaltung der externen Speichermedien obliegt dem Betriebssystem, hier speziell der Dateisystemverwaltungskomponente. Deren Aufgabe ist das

- Verbergen der speziellen Geräteeigenschaften,
- die Verwaltung von freiem und belegtem Speicher,
- die Abbildung der Datenbankblöcke auf die Speichereinheiten der jeweiligen Medien und
- die Kontrolle des Datentransfers zwischen Datenbankpuffer und Datenträgern.

Auf Seiten von DBMS werden unter Nutzung der Funktionalitäten des jeweiligen Dateisystems u.a. die folgenden Erweiterungen der Funktionalitäten realisiert:

- verzögerte Einbringung von Änderungen zur Unterstützung von Rollback- und Recovery-Operationen
- Einführung von Segmenten als Einheiten für Sperren, Wiederherstellung oder Zugriffskontrolle

Dazu müssen u.a. die folgenden Aufgaben gelöst werden:

- Abbildung der verschiedenen Segmenttypen (Daten- und Index-Segmente, temporäre Segmente, Log-Segmente) auf Dateien (die möglichst eine direkte Blockadressierung erlauben)
- Realisierung von Strategien zur verzögerten Einbringung von Änderungen (Schattenspeicherkonzept, differential file, ...)
- Vorbereiten von Lese- und Schreibanforderungen
- Verwalten von Pufferbereichen im Hauptspeicher
- Bereitstellen/Freigeben von Datenbankobjekten im Datenbankpuffer

- Ersetzungsverfahren zur optimierten Nutzung von datenbankspezifischen Referenzlokalitäten

Datenbank-Management-Systeme verwenden eigene Puffer, weil die alleinige Nutzung des Puffers des Dateisystems aus verschiedenen Gründen ungünstig ist.

- Wenn Daten direkt aus dem Puffer des Betriebssystems geholt werden müssten, wäre jeweils ein Kontextwechsel nötig, da Systemaufrufe ausgeführt werden würden.
- Die Pufferverwaltung des Dateisystems wird den verschiedenen Zugriffsmustern von DBMS (sequenzielle Zugriffe, Zugriffe über Indexe usw.) u.U. nicht gerecht. Lange sequenzielle Scans könnten z.B. den Puffer „putzen".
- Das vorausschauende Lesen (Prefetching) muss u.U. anders organisiert werden, da Daten, die in der Datenbank fortlaufend abgelegt werden, nicht zwangsläufig auch so auf den Datenträgern gespeichert sein müssen.
- Wenn vom DBMS das sofortige Schreiben eines Blocks ausgelöst wird, sollte der Block auch wirklich sofort auf den entsprechenden Datenträger geschrieben werden (z.B. beim Schreiben von Log-Einträgen).

Eine wichtige Aufgabe der Pufferverwaltung eines DBMS ist die Realisierung einer geeigneten Ersetzungsstrategie für die verfügbaren Pufferrahmen. Gängige Strategien werden im Folgenden kurz beschrieben.

- *LRU* (Least Recently Used) – Bei dieser Ersetzungsstrategie wird für jeden Rahmen eine Information mitgeführt, wann zum letzten Mal auf eine Seite zugegriffen wurde. Es wird die Seite ausgelagert, auf die am längsten nicht zugegriffen wurde.
- *LFU* (Least Frequently Used) - Bei dieser Ersetzungsstrategie wird für jeden Rahmen ein Referenzzähler mitgeführt, der bei jedem Zugriff erhöht wird. Es wird die Seite mit dem niedrigsten Referenzzählerwert ausgelagert.
- *LRU-K* – Bei dieser Strategie wird für die letzten K-Zugriffe noch eine Information mitgeführt, wann der Zugriff erfolgt ist. Bei der Suche nach dem Rahmen, dessen Inhalt ersetzt werden soll, werden die Zeitabstände zwischen den letzten K-Zugriffen (Referenzdichte) mit in die Entscheidung einbezogen.
- *Clock-Algorithmus* – Für alle Rahmen wird ein Bit geführt, ob der Rahmen während des letzten Durchlaufzyklus benutzt wurde. Zyklisch werden die Bits der Rahmen auf 0 gesetzt. Wird ein Rahmen benutzt, wird das Bit auf 1 gesetzt. Wird jetzt ein Rahmen benötigt, so wird eine Seite ersetzt, deren Bit auf 0 steht.
- *TC-Algorithmus* (Oracle) – Die Verwaltung der Rahmen erfolgt in einem sog. LRU-Stack. Ein Rahmen, dessen Inhalt eben ersetzt wurde, wird in der Mitte des Stacks eingefügt. Je Rahmen wird ein sog. *Touch Counter* (TC) geführt, der bei jedem Zugriff erhöht wird. Der LRU-Stack hat ein „kaltes" und ein „heißes" Ende. Die Inhalte der Rahmen am kalten Ende werden ersetzt. Wird bei der Suche

nach freien Rahmen bzw. von einem Page Cleaner[1] ein Rahmen gefunden, dessen TC einen Schwellwert übersteigt, so wird der Rahmen an das heiße Ende des Stacks verschoben und sein TC auf 0 gesetzt. Der Rahmen „altert“ und rutscht langsam in Richtung des kalten Endes. Wird wieder auf den Inhalt des Rahmens zugegriffen, so wird der TC bei jedem Zugriff wieder erhöht.

6.3 Sekundärspeicherorganisation bei DBMS

Die prinzipielle Speicherorganisation von gängigen DBMS-Produkten ähnelt einander weitgehend. Da von verschiedenen Produkten aber oft unterschiedliche Bezeichnungen verwendet werden, sollen hier zunächst einheitliche Bezeichnungen eingeführt werden. Zur Speicherorganisation gehören Verwaltungs- und Speichereinheiten (Elemente), die auf unterschiedlichen Ebenen der in *Abbildung 8* dargestellten Schichtenarchitektur angesiedelt sind. Die Verwaltungseinheiten dienen als logische Container für die Speichereinheiten. Die Eigenschaften der Verwaltungseinheiten können auf einer gegenüber der SQL-Norm (die dieses ausklammert) erweiterten DDL-Ebene verändert werden. Solche Erweiterungen können im Rahmen von Datendefinitionsanweisungen benutzt werden, um beispielsweise die Platzierung von Datenbankobjekten zu beeinflussen. *Abbildung 40* zeigt den Zusammenhang der einzelnen Elemente.

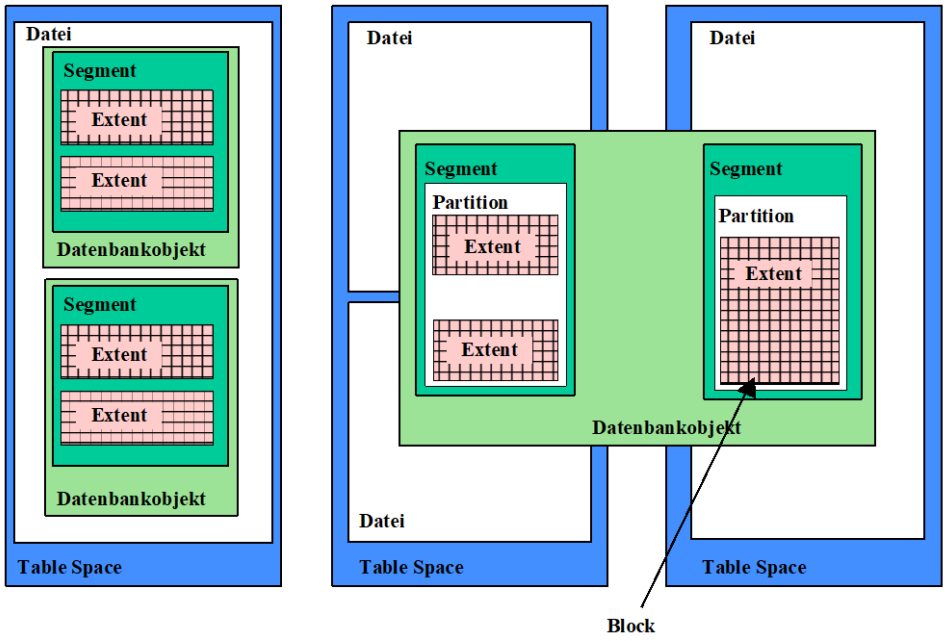


Abbildung 40: Sekundärspeicherorganisation im Überblick

Zu den *Speichereinheiten* zählen Dateien, Extents und Blöcke. Die eigentliche Speicherung von Daten erfolgt auf externen Speichermedien (typischerweise Plattenspeicher). Die Verbindung zu diesen externen Speichermedien wird bei modernen Rechnerarchitekturen und Betriebssystemen über eine Dateischnittstelle

[1] Page Cleaner sind die Prozesse, die für das asynchrone Schreiben geänderter Pufferinhalte zuständig sind.

realisiert. Die Organisation dieser *Dateien* unterhalb der Schnittstelle zum Betriebssystem ist unterschiedlich. So kann eine Datei vom Dateisystem des Betriebssystems verwaltet werden. Das ist dann für die Reservierung oder Freigabe von Speicher verantwortlich und führt i.d.R. auch eine Zwischenpufferung im Arbeitsspeicher durch. Durch Reservierung und Freigabe von Speicher verschiedener Dateien kann es im Laufe der Zeit dazu kommen, dass der einer Datei zugeordnete Externspeicher in kleinen Stücken (Fragmenten) über den physischen Datenträger verstreut ist. Dies erfordert beim Lesen der Datei, gegenüber einer physisch fortlaufenden Speicherung, eine erhöhte Anzahl notwendiger Positionierungsoperationen der Lese-/Schreibköpfe, was zu Performance-Einbußen führt. Zur Verringerung der Gefahr solcher Fragmentierungen wird typischerweise bereits bei der Erzeugung von Datenbankdateien ein größerer Speicherbereich soweit möglich zusammenhängend reserviert. Eine andere Form der Dateischnittstelle stellen die sog. *Raw Devices* dar. Dahinter verbirgt sich jeweils ein zusammenhängender Speicherbereich, auf den über eine vom Betriebssystem zur Verfügung gestellte Dateischnittstelle direkt, auch unter Umgehung der Dateipufferung des Betriebssystems, zugegriffen werden kann. Durch hoch entwickelte Hard- und Softwarekomponenten (RAID-Systeme, Volume Manager usw.) kann der für ein Dateisystem oder als Raw Device verfügbare Speicher auf unterschiedliche Art und Weise aus Teilen eines oder mehrerer Datenträger zusammengestellt werden. Die unterschiedlichen Dateiorganisationsformen bleiben dem DBMS allerdings größtenteils verborgen. Sie liegen „unterhalb“ des Datenbank-Management-Systems. Auch die Verantwortung für innerhalb der Dateien eventuell vorhandene (aus Sicht des DBMS externe) Fragmentierungen liegt außerhalb des DBMS.

Lese- und Schreiboperationen (I/O-Operationen) werden über die Dateischnittstelle aus Effizienzgründen nicht byte-weise, sondern in *Blöcken* oder Sequenzen von Blöcken abgewickelt. Dabei entspricht die Größe eines Blocks typischerweise der Speichermenge, die mit einer I/O-Operation von einem Datenträger gelesen bzw. auf diesen geschrieben werden kann oder einem Vielfachen davon. Die wesentlichen Teile eines Blocks sind

- der Block Header, der allgemeine Verwaltungsinformationen enthält,
- eventuell eine Slot-Liste (bzw. Row Directory), die Verweise auf die einzelnen im Block gespeicherten Einträge enthält und
- der Bereich, der zur Speicherung der eigentlichen Daten bzw. Indexeinträge zur Verfügung steht.

Um Fragmentierungen zu verringern, wird der Speicherplatz von Datenbankobjekten bei Bedarf nicht blockweise, sondern in Form von *Extents* reserviert. Ein Extent ist eine Menge physisch beieinander liegender Blöcke innerhalb einer Datei. Die Extent-Größe kann entweder vom DBMS vorgegeben sein oder sie gehört zu den (spezifizierbaren) Eigenschaften des entsprechenden Datenbankobjekts oder des das Objekt enthaltenden Table Space.

Die einem Datenbankobjekt zugeordneten Extents bilden ein *Segment*. Liegen in einem Table Space mehrere Segmente, die fortlaufend erweitert werden, so ist es möglich, dass die zu einem Segment gehörenden Extents nicht physisch fortlaufend reserviert werden können (*Abbildung 41*). Damit erhöht sich der Aufwand für Positionierungsoperationen der Lese-/Schreibköpfe jeweils beim Übergang zum nächsten Extent. Eine wohlüberlegte und datenbankobjektbezogene Festlegung von Extent-Größen kann solchen Fragmentierungen entgegenwirken, erschwert aber auch die Wiederverwendung freigegebener Extents für andere Datenbankobjekte. Wegen der typischerweise geringen Anzahl von Extents, die das Segment eines Datenbankobjekts umfasst, ist der Einfluss der Segmentfragmentierung auf die Systemleistung aber wohl eher gering. Ausgefeilte Techniken zur internen Verwaltung physischer Speicherungsstrukturen können auch eine effiziente Wiederverwendung von Freispeicherfragmenten unterschiedlicher Größe ermöglichen.

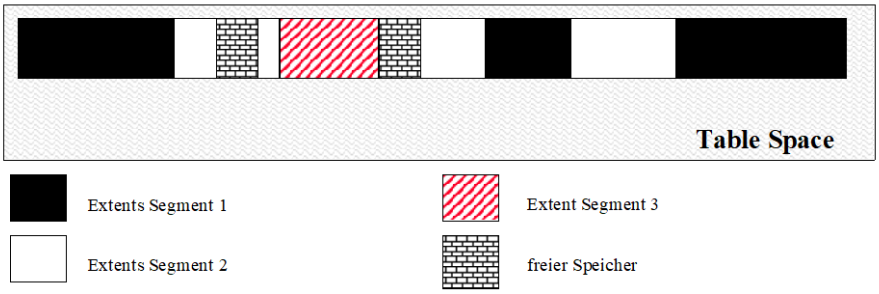


Abbildung 41: Segmentfragmentierung

Verwaltungseinheiten sind Table Spaces, Datenbankobjekte und Partitionen. Der gesamte Speicherplatz einer Datenbank wird in einem oder mehreren *Table Spaces* verwaltet. Ein Table Space umfasst eine oder mehrere Dateien. Table Spaces stellen um Dateien erweiterbare Container dar, die Datenbankobjekte enthalten können. Wird einem Table Space jeweils nur eine Datei zugeordnet, so kann darüber die Platzierung von Datenbankobjekten auf den Datenträgern gesteuert werden (etwa gezielt auf unterschiedliche Datenträger oder auch gemeinsam auf einem Datenträger).

Innerhalb von Table Spaces können *Datenbankobjekte* abgelegt werden. Zu den Datenbankobjekten, die hier von Bedeutung sind, zählen Tabellen, Indexe und Cluster. Während in Tabellen nur Daten einer Relation (im Sinne des relationalen Datenmodells) gespeichert werden, können Cluster die Daten mehrerer logisch zusammengehöriger Relationen enthalten. Auf das Cluster-Konzept wird in *Abschnitt 6.4* noch detaillierter eingegangen.

Tabellen und Indexe können bei Bedarf nach bestimmten Kriterien weiter unterteilt (partitioniert) werden. Diese Unterbereiche werden als *Partitionen* bezeichnet. Die Partitionen werden Table Spaces zugeordnet, in denen für jede Partition ein eigenes Segment angelegt wird.

Durch die von DBMS zur Datenspeicherung verwendete Speicherorganisation sind in der Regel gewisse Grenzen (Schwellwerte) bezüglich der maximalen Größe von Segmenten bzw. der maximalen Anzahl Extents, die ein Segment enthalten kann, gesetzt. Weiterhin ist es möglich, dass der in einem Table Space zur Verfügung stehende Speicherplatz erschöpft ist und der Table Space vor weiteren Operationen erst erweitert werden muss.

Auch auf Betriebssystemebene existieren solche Grenzen (bspw. maximale Dateigrößen). DBMS- und Betriebssystemhersteller haben in den vergangenen Jahren intensiv daran gearbeitet, solche Grenzen zu überwinden oder die Grenzen so weit zu verschieben, dass sie in der Praxis kaum erreicht werden können. Allerdings steht u.U. die Frage nach der Effizienz solcher Implementierungen. Oft ist der Verwaltungsaufwand für selbsterweiternde Strukturen mit potenziell „unendlichem" Fassungsvermögen wesentlich höher als bei Strukturen, bei denen klare Grenzen gesetzt sind. Dort ist aber meist wieder der Administrationsaufwand höher.

Ist z.B. die Anzahl der Extents, die für eine Tabelle reserviert werden können, begrenzt, weil nur ein Block für Verwaltungsdaten von Extents zur Verfügung steht, so muss der DBA, wenn die Grenze erreicht wird, mit einer Datenbankreorganisation dafür sorgen, dass zukünftig größere Extents reserviert werden. Sonst wäre beim weiteren Anwachsen der Tabelle der Datenbankbetrieb gefährdet. Ist die Zahl der Extents nicht begrenzt, z.B. weil weitere Verwaltungsblöcke reserviert und in einer Kette verwaltet werden können, so tritt die akute Gefährdung des Datenbankbetriebs nicht ein. Dafür ist hier ein höherer Aufwand beim Verwalten und auch beim Auffinden der Extents durch das DBMS nötig. Dies wirkt sich negativ auf die Systemleistung aus.

Welche Grenzen existieren und wo die jeweilige Grenze genau liegt, ist systemabhängig und kann allgemeingültig kaum erfasst werden. Hier muss der DBA die in der speziellen Systemkonfiguration auftretenden Grenzen ermitteln. Zusätzlich sind geeignete Methoden anzuwenden, um Entwicklungen, die zum Erreichen solcher Grenzen führen können, rechtzeitig zu erkennen und zu überwachen. Dabei wirken als wesentliche Einflussfaktoren die noch verbleibende Zeit, bis der normale Datenbankbetrieb auf Grund der erreichten Grenzwerte nicht mehr aufrecht erhalten werden kann und die Einschränkungen im normalen Datenbankbetrieb, die die Reorganisation verursacht. Es muss ein Kompromiss gefunden werden, der einerseits die rechtzeitige Reorganisation sicherstellt und andererseits die Verfügbarkeitsanforderungen der Anwender weitestgehend berücksichtigt.

6.4 Speicher- und Zugriffskonzepte

6.4.1 Tabellen und Indexe

Für rein („klassisch") relationale Systeme ist die Beschreibung der logischen Datenstrukturen recht einfach. Daten werden in einfacher Tabellenform, ohne mengenwertige oder zusammengesetzte Attribute dargestellt. Weder die Reihenfolge der Attribute noch die der Tupel einer *Tabelle* sind vorgeschrieben. Eine Tabelle kann daher auf eine ungeordnete Folge von Datensätzen abgebildet werden. Um

Zugriffe auf einzelne Sätze bzw. kleinere Satzmengen zu beschleunigen, können häufig für Suchoperationen verwendete Attribute bzw. Attributkombinationen (*Suchschlüssel*) indexiert werden. Über den *Index* wird eine Zuordnung zwischen Suchschlüsselwert und den Speicherorten der den Suchschlüsselwert enthaltenden Datensätze vorgenommen. Damit werden zusätzliche Zugriffspfade geschaffen. Im Zusammenhang mit der Indexierung bieten Datenbank-Management-Systeme die Möglichkeit, Sätze nach dem Suchschlüssel eines Index sortiert (geclustert) zu speichern. Ein solcher Index wird häufig auch als *Clustered Index* bezeichnet. Durch die *wertebasierte Clusterung* kann insbesondere für Bereichsanfragen und Anfragen, bei denen das Ergebnis nach dem Ordnungskriterium des Index sortiert benötigt wird, der notwendige Such- und Sortieraufwand verringert werden. Deshalb sollte jeweils der Schlüssel zur Clusterung ausgewählt werden, der bezüglich der Tabelle zur Ausführung der genannten Operationen die größte Systemlast – und damit den größten Nutzen durch die Clusterung – erwarten lässt. Abhängig von der internen Realisierung solcher Indexe (z.B. bei der Invertierung von als Heap organisierten Datenbereichen) kann die Clusterung im Laufe der Zeit durch Einfüge- und Änderungsoperationen verloren gehen.

DBMS-Produkte bilden objektorientierte Erweiterungen intern meist noch auf rein relationale Strukturen ab. Dadurch werden Redundanzen und Anomalien bei Update-Operationen vermieden. Allerdings müssen viele Verbundoperationen ausgeführt werden, wenn bspw. alle Daten eines komplexen Objekts benötigt werden. Eine andere Variante ist die Speicherung der Daten komplexer Objekte im Binärformat, dessen interne Struktur dem DBMS aber verborgen bleibt.

6.4.2 Horizontale Partitionierung von Tabellen

Zur Beschleunigung von gegen sehr große Tabellen gerichteten Anfragen und zur Verbesserung der Administrierbarkeit bieten einige DBMS-Produkte (z.B. Oracle, DB2, Informix, MS SQL Server) Konzepte zur *horizontalen Partitionierung* von Tabellen und Indexen an. Bei dieser Partitionierungsform wird eine Tabelle horizontal und vollständig in disjunkte Teiltabellen (*Partitionen*) unterteilt, die dann gezielt physischen Bereichen der Datenbank zugeordnet werden (*Abbildung 42*). Die einzelnen Partitionen können dann typischerweise einzeln administriert werden, und Wartungsaufgaben können zielgerichteter und auf kleinere Einheiten beschränkt („lokaler“) durchgeführt werden.

Wie die Aufteilung erfolgt, wird durch ein *Partitionierungsschema* unter Verwendung einer bestimmten *Partitionierungsstrategie* definiert. Dazu können prinzipiell die folgenden Strategien angewendet werden, die allerdings nicht von allen DBMS-Produkten in jener Breite angeboten werden:

- Bei der *Round-Robin-Strategie* werden die Tupel einer Tabelle bei Einfügung der Reihe nach, also gleichmäßig, auf die einzelnen Partitionen verteilt. Dadurch kann beim späteren lesenden Zugriff eine gute Lastbalancierung erreicht werden. Weiterhin wird eine Beschleunigung von sequenziellen Suchoperationen möglich, indem die einzelnen Partitionen parallel sequenziell durchsucht werden und am

Ende die Ergebnisse der (Teil-)Anfragen zusammengeführt werden. Punkt- oder Bereichsanfragen profitieren von dieser Strategie jedoch nicht.

- Bei der *Hash-Strategie* wird aus dem Wert eines oder mehrerer Attribute eines Tupels ein Hash-Wert berechnet, der die Zuordnung zu einer Partition bestimmt. Neben der Möglichkeit, sequenzielle Suchoperationen zu parallelisieren, profitieren bestimmte Punktanfragen von dieser Partitionierungsstrategie.
- Bei *ausdrucksbasierten Partitionierungsstrategien* bestehen die größten Freiheiten. So können Tupel, z.B. bereichsweise nach dem Partitionierungsschlüssel, in die verschiedenen Partitionen einer Tabelle eingeordnet (wertebereichsbezogen geclustert) werden. Sequenzielles Suchen und Bereichsanfragen profitieren von dieser Partitionierungsform neben der Clusterung auch dadurch, dass Partitionen, die definitiv keine Tupel der Treffermenge enthalten können, u.U. bei der Suche ausgeschlossen werden können. Dies führt dann zu einer erheblichen Verringerung der zu durchsuchenden Datenmenge.

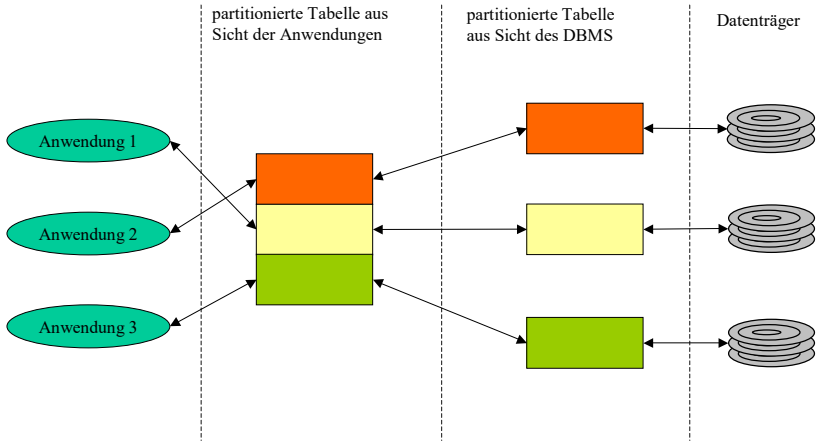


Abbildung 42: Partitionierung von Tabellen

Gegenüber Anwendungsprogrammen wird die Partitionierung verborgen (Transparenz). DBMS behandeln sie bezüglich der Speicherung i.d.R. wie eigenständige Tabellen. Jeder Partition wird ein eigenes Segment zugeordnet. Zur Ausnutzung aller Vorteile, die hash- und ausdrucksbasierte Partitionierungsstrategien bieten, ist es notwendig, dass die an der Anfrageausführung beteiligten Komponenten Kenntnis über die Partitionierungsstrategie besitzen und dass die Selektionsbedingungen der Anfragen zu diesem Schema „passen“. Bei Entwurf und Überarbeitung von Partitionierungsschemata müssen daher Informationen oder Annahmen über die gegen die entsprechenden Tabellen gerichteten Anweisungen (Workload) berücksichtigt werden.

Die Anwendung von Partitionierungsmechanismen kann zur Verbesserung der Administrierbarkeit beitragen. Administrationsmaßnahmen können zielgerichtet auf kleinere Einheiten angewendet werden. In gewisser Hinsicht kann Partitionierung auch zu einer Erhöhung der Datenverfügbarkeit beitragen. Fällt ein Teil einer nicht partitionierten Tabelle, die innerhalb eines Table Space auf mehrere Dateien auf

unterschiedlichen Datenträgern verteilt ist, durch einen Datenträgerfehler aus, so steht die gesamte Tabelle nicht mehr zur Verfügung. Für partitionierte Tabellen kann u.U. die Verarbeitung auf den noch verfügbaren Partitionen weiter ausgeführt werden. Einige DBMS-Produkte auch können so konfiguriert werden, dass durch Datenträgerfehler oder die Durchführung von Administrationsmaßnahmen ausgefallene Partitionen „übersprungen“ werden. Allerdings sollte vorab genau geprüft werden, ob die ggf. damit verbundenen Fehler bei der Erstellung der Treffermengen von Such- und Update-Operationen toleriert werden können (was etwa bei statistischen Auswertungen der Fall sein kann).

6.4.3 Tabellenübergreifende Clusterung

Vorrangig zur Unterstützung von Verbundoperationen (hier Equi Joins) bietet das DBMS-Produkt Oracle (als eines von relativ wenigen Produkten) die Möglichkeit der *tabellenübergreifenden Clusterung* von Daten. Hier können Tupel unterschiedlicher Tabellen, die über Schlüssel (*Cluster-Schlüssel*) in Beziehung (typischerweise 1:N) zueinander stehen, gemeinsam in einem Datenblock gespeichert werden. Der Zugriff auf die Tupel erfolgt über einen gemeinsamen Zugriffspfad, der als „normaler“ B*-Baum-Index oder als Hash-Index realisiert sein kann. Der Speicherbereich, der die zu einem Cluster-Schlüsselwert bzw. Hash-Wert gehörenden Tupel aufnimmt, wird als *Bucket* bezeichnet. Die in einem Bucket gespeicherten Tupel werden auch als *Cluster-Gruppe* bezeichnet. *Abbildung 43* zeigt ein Beispiel, bei dem jeweils die persönlichen Daten eines Mobilfunkteilnehmers und seine Verbindungsdaten gemeinsam in einem Bucket gespeichert werden.

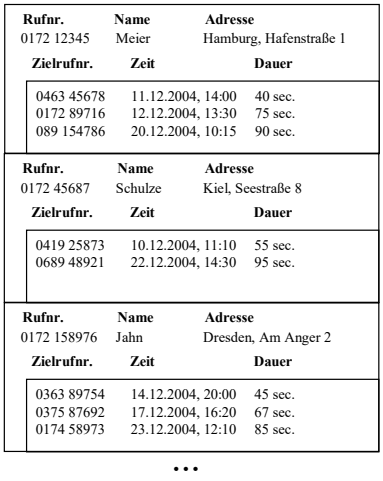


Abbildung 43: Tabellenübergreifende Clusterung von Tupeln

Die Zahl der Tabellen, die an einer Cluster-Definition beteiligt sein können und deren Daten in einem Cluster abgespeichert werden können, ist theoretisch nicht begrenzt.

Anhand der in *Abbildung 44* dargestellten Beispiele sollen kurz Einsatzgebiete und Grenzen der Cluster-Bildung erläutert werden. Primärschlüsselattribute sind unterstrichen, Fremdschlüssel kursiv und Cluster-Schlüssel fett dargestellt.

Beispiel 1:

```
Filiale(Filialnr,PLZ,Ort,Straße,...);
Mitarbeiter(Manr,Name, Vorname,...,Filialnr);
Kunde(Kdnr,Name,Vorname,...,Filialnr);
```

Beispiel 2:

```
Filiale(Filialnr,PLZ,Ort,Straße,...);
Mitarbeiter(Manr,Name,Vorname,PLZ,Wohnort,...,Filialnr);
Kunde(Kdnr,Name,Vorname,PLZ,Wohnort,...,Filialnr);
```

Beispiel 3:

```
Flug(Flugnr,Start,Ziel,...);
Passagier(Panr,Name,Vorname, ...);
Buchung(Buchungsnr,Flugnr,Panr,Platznr,...);
```

Abbildung 44: Beispiele zur Cluster-Bildung

Die Beispiele 1 und 2 zeigen einen kurzen Ausschnitt aus einer Unternehmensdatenbank, die Daten von Filialen und deren Mitarbeitern sowie die Daten der von den einzelnen Filialen betreuten Kunden enthält. Die Filialnummer, die in allen Tabellen enthalten ist, ist ein typischer Kandidat für einen Cluster-Schlüssel. Die Daten der Mitarbeiter einer Filiale und die Daten der Kunden der Filiale würden dicht bei den Filialdaten gespeichert werden. Die Clusterung kann aber auch für Nicht-Schlüsselattribute, also für beliebige Attribute, vorgenommen werden.

In Beispiel 2 dienen die in den Tabellen in Form von Sekundärschlüsseln enthaltenen Ortsangaben zur Clusterung. Hier werden die Daten von Filialen, Kunden und Mitarbeitern, die am gleichen Ort ansässig sind, physisch dicht beieinander gespeichert. Die Objekte der im Cluster gespeicherten Tabellen müssen also nicht zwangsläufig hierarchisch zueinander strukturiert sein. M:N-Beziehungen können über den Cluster-Schlüssel allerdings nicht dargestellt werden. Beispiel 3 zeigt einen Ausschnitt aus einer Datenbank zur Buchung von Flügen. Die Tabelle Buchung dient dabei zum Ausdrücken der M:N-Beziehung zwischen Passagieren und Flügen über die beiden Fremdschlüsselattribute `Panr` und `Flugnr`. Es ist auch möglich, einen Cluster-Schlüssel aus einer Attributkombination zu bilden (mehrattributige Cluster-Schlüssel). Allerdings müssen dann alle Tabellen, die im Cluster untergebracht werden sollen, diese Attributkombination vollständig enthalten und nicht nur Teile des Schlüssels (wie bei M:N üblich und in Beispiel 3 gezeigt). Durch diesen gemeinsamen Cluster-Schlüssel ist die Anzahl der Hierarchieebenen, über die Datenobjekte geclustert werden können, auf zwei begrenzt (es gibt sozusagen keine Möglichkeit des „Clusterns innerhalb des Clusters“).

Die Vorteile der tabellenübergreifenden Clusterung sind dann besonders ausgeprägt, wenn Zugriffsoperationen hauptsächlich über den Cluster-Schlüssel erfolgen und wenn die logisch in Beziehung zueinander stehenden Daten auch häufig zusammen

benötigt werden. Hier kann die physisch zusammenliegende Speicherung der Tupel die Anzahl der für den Verbund notwendigen Blockzugriffe gegenüber der in relationalen Systemen sonst üblichen getrennten Speicherung deutlich vermindern. Die Vermischung der Tupel unterschiedlicher Tabellen führt bei auf einzelne Tabellen des Clusters angewendete sequenzielle Suchoperationen aber auch dazu, dass meist deutlich mehr Datenblöcke durchsucht werden müssen, als bei der getrennten Speicherung der Tabellen. Es gilt die bekannte Eigenschaft: Man kann nur nach einem Kriterium clustern (entweder *innerhalb* der Tabelle oder *tabellenübergreifend*). In die Entscheidung darüber, ob eine tabellenübergreifende Clusterung sinnvoll ist, müssen, wie z.B. bei Entscheidungen über die Partitionierung von Tabellen auch, Informationen oder Annahmen über die gegen die zu clusternden Tabellen gerichteten Anweisungen (Workload-Charakteristika) berücksichtigt werden.

6.5 Interne Strukturen zur Datenspeicherung und Degenerierungen

6.5.1 Datenspeicherung als Heap

Die Speicherung von Informationen in Datenbanken erfolgt in Blöcken, aus denen komplexere interne Speicherungsstrukturen gebildet werden. Die Speicherung von Tupeln und Indexinformationen erfolgt in Form von Sätzen und Einträgen, die in Blöcken abgelegt werden.

Eine häufig von relationalen Systemen verwendete Struktur zur Speicherung von Daten ist die **Heap-Struktur**. Die einzelnen Sätze werden fortlaufend in Zugangsreihenfolge untergebracht. Zur Speicherung wird ein Segment angelegt, das bei Bedarf um weitere Extents vergrößert wird (vgl. Abschnitt 6.3). Wird durch Löschoperationen oder tupelverkürzende Änderungsoperationen Speicher frei, so kann (oder soll) dieser meist nicht unmittelbar wieder belegt werden. Das führt sukzessive zu einer Vermischung von freiem und belegtem Speicher innerhalb der Blöcke. Der frei gewordene Speicher (*eingestreuter Freiplatz)* liegt meist in so kleinen Einheiten vor, dass er von der Freispeicherverwaltung häufig nicht sinnvoll wieder vergeben werden kann, bzw. dieser zunächst nicht wieder zur Verfügung gestellt wird. Um die Entwicklung von Freiplatzfragmenten zu verlangsamen, legt bspw. das DBMS-Produkt Oracle für jedes Segment Freispeicherlisten an, in die Blöcke eingeordnet werden, deren Belegungsgrad nach einer nahezu vollen Belegung unter eine definierte Schwelle (definiert über den Parameter `PCTUSED`) sinkt. Bei Einfügeoperationen wird versucht, Sätze zunächst in einem in einer Freiliste befindlichen Block unterzubringen, bevor neue Blöcke belegt werden. Mit Einführung des *Automatic segment-space management* wurden unter Oracle Bitmap-Listen eingeführt, die u.U. die Konfiguration von Freispeicherlisten und Parametern wie `PCTFREE` und `PCTUSED` überflüssig machen [Ora03a]. Eingestreuter Freiplatz führt zu einer *Verschlechterung der Speicherauslastung* und zur Erhöhung des I/O-Aufwands beim sequenziellen Suchen, da die Freiplatzfragmente mitgelesen werden müssen. Bei Zugriffen über Indexe hat in Datenbereiche eingestreuter Freiplatz kaum Auswirkungen. Allerdings ist eine vollständige Vermeidung eingestreuten Freiplatzes i.d.R. auch nicht anzustreben. Bei Einfüge- und Änderungsoperationen

kann eine „Freispeicherreserve“ in den Datenblöcken durchaus positive Auswirkungen haben, die nachfolgend noch näher erläutert werden.

Wenn die Länge von Sätzen die Größe des in den Blöcken zur Datenspeicherung verfügbaren Platzes übersteigt und das DBMS dies zulässt, müssen die Sätze auf mehrere Blöcke verteilt (fragmentiert) werden. Für die *Fragmentierung von Sätzen* sind unterschiedliche Vorgehensweisen denkbar. Einige davon soll das folgende (zugegebenermaßen konstruierte) einfache Beispiel verdeutlichen.

Wenn ein Satz (inkl. des Verweises auf das nachfolgende Fragment) das 1.75-fache des in einem Block zur Datenaufnahme verfügbaren Speichers belegt, so könnten vier Sätze (S1 ... S4), wie im folgenden Bild gezeigt, in sieben oder acht Speicherblöcken (B1 ... B7 bzw. B8) untergebracht werden (*Abbildung 45*).

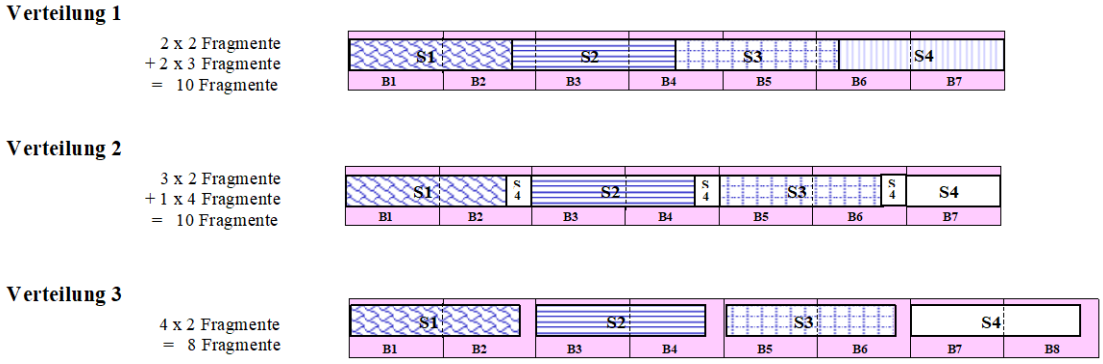


Abbildung 45: Verteilungsvarianten von Sätzen

Tabelle 3 zeigt einen Vergleich der dargestellten Varianten bezüglich benötigtem Speicherplatz in Blöcken, der Anzahl anfallender Blockzugriffe beim sequenziellen Suchen nach den vier Sätzen und der Anzahl Zugriffe auf Datenblöcke bei Punktanfragen unter Nutzung von Indexen, wobei eine Gleichverteilung der Zugriffe auf die Sätze angenommen wird.

Speicherplatz in Blöcken	Aufwand für sequenzielles Suchen	Aufwand für Punktzugriffe
7	7	2,5
7	7 bzw. 9	2,5
8	8	2

Tabelle 3: Vergleich der Varianten

Bei den ersten beiden Verteilungen wird der zur Verfügung stehende Speicher optimal ausgenutzt. Diese optimale Ausnutzung des Speichers führt allerdings dazu, dass gegenüber der dritten Verteilung eine höhere Anzahl Satzfragmente entsteht. Bei *Verteilung 1* werden zwei Sätze (sozusagen „ohne echte Not“) in drei anstelle der mindestens notwendigen zwei Fragmente aufgeteilt. Diese Erhöhung der Fragmentzahl wird beim sequenziellen Suchen ohne größere Auswirkungen bleiben. Voraussetzung dafür ist, dass es gelingt, die Verteilung bei Änderungs- und

Löschoperationen aufrecht zu erhalten. Dies ist allerdings mit vertretbarem Aufwand kaum zu realisieren.

Bei *Verteilung 2* entspricht die Gesamtzahl der Fragmente der von Verteilung 1. Die Sätze werden nur anders aufgeteilt. Beim sequenziellen Suchen verursacht die Fragmentierung von Satz S4 hier entweder eine deutliche Erhöhung der Anzahl Blockzugriffe (wenn immer erst alle Fragmente eines Satzes gelesen werden, bevor der nächste Satz verarbeitet wird) oder es muss ein hoher Aufwand für das Zwischenspeichern und „Merken" einzelner Satzfragmente betrieben werden.

Bei *Verteilung 3* wird mehr Speicher benötigt als bei den ersten beiden Verteilungen. Der Aufwand für die sequenzielle Verarbeitung ist höher als bei Verteilung 1. Bei gleich verteilten Zugriffen über Punktanfragen sind aber jeweils nur zwei Zugriffe auf Datenblöcke nötig.

Als Zugriffsmuster für eine Tabelle muss i.d.R. ein Mix aus sequenziellem Suchen und Punktanfragen (und Bereichsanfragen unter Nutzung von Indexen) angenommen werden. Auch der Aufwand für Einfüge-, Lösch- und Änderungsoperationen muss berücksichtigt werden. Deshalb verteilen Datenbank-Management-Systeme üblicherweise Sätze auf jeweils möglichst wenige Blöcke (ähnlich Verteilung 3). Das heißt, je nach Satzlänge werden zunächst ein Block oder mehrere Blöcke vollständig gefüllt. Ein verbleibendes Restfragment wird in einem weiteren Block, der dann nicht vollständig gefüllt ist, gespeichert. Der dadurch entstehende eingestreute Freiplatz wird toleriert. Um den Freiplatzanteil möglichst klein zu halten, speichern manche DBMS-Produkte mehrere Restfragmente gemeinsam in einem Block, wenn das deren Länge erlaubt.

6.5.2 Indexierung bei getrennter Speicherung von Daten und Indexen

Zur Beschleunigung von Zugriffen auf einzelne Sätze bzw. kleinere Satzmengen werden durch **Indexierung** weitere Zugriffspfade geschaffen. Die Indexierung erfolgt dabei über einen Indexierungsschlüssel, der aus den Werten eines Felds bzw. einer Kombination von Feldern gebildet wird. Über den Index wird eine Zuordnung von Werten des Indexierungsschlüssels zu den Speicherorten der Sätze vorgenommen, die die entsprechenden Schlüsselwerte enthalten. Diese Zuordnung erfolgt über Wertepaare, die aus dem jeweiligen Wert des Indexierungsschlüssels und Verweisen auf die Speicherorte der die Schlüsselwerte enthaltenden Sätze gebildet werden. Dabei können logische Verweise verwendet werden, die z.B. dem Indexierungsschlüsselwert den korrespondierenden Schlüsselwert eines anderen (primären) Index zuordnen[2] oder physische Verweise (Zeiger), die oftmals als Tupelidentifikatoren (*TID*) oder Record Identifier (*RID*) bzw. Row Identifier (*ROWID*) bezeichnet werden. Tupelidentifikatoren enthalten typischerweise einen Verweis auf den physischen Speicherort eines Datensatzes (u.a. die Blocknummer). Die Speicherung der Indexe erfolgt meist von den Daten getrennt in jeweils eigenen Segmenten. Als Organisationsform für Indexe werden größtenteils Varianten von B-

[2] Derartige Verweiskonzepte werden bspw. bei indexorganisierten Tabellen, auf die noch eingegangen wird, verwendet.

Bäumen (B^+-Bäume bzw. B^*-Bäume) verwendet. Bei Tabellen, deren Daten als Heap gespeichert werden, werden auf der Blattebene des Indexbaums Verweise auf die zu den Schlüsselwerten gehörenden Datensätze gespeichert, da die Datensätze selbst hier außerhalb des Baums abgelegt werden. Beim Zugriff auf einen Datensatz (Index Lookup) wird zunächst der Indexbaum von der Wurzel beginnend durchsucht. Anschließend wird auf den Datenblock zugegriffen, auf den der Zeiger in der Blattebene verweist (*Abbildung 46*).

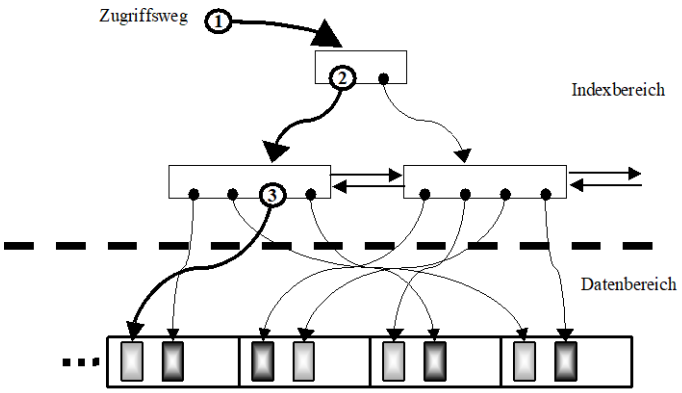


Abbildung 46: Blockzugriffe bei einem Index Lookup

Durch Änderungsoperationen können Datensätze verlängert werden. Dies kann dazu führen, dass die betroffenen Datensätze nicht mehr in den ursprünglichen Datenblöcken gespeichert werden können und komplett in andere Datenblöcke verschoben werden müssen. Bei der Verwendung physischer Verweise müssten diese entsprechend aktualisiert werden. Dies ist u.U. mit einem erheblichen Aufwand verbunden, besonders wenn die Tabelle nach mehreren Kriterien indexiert ist. Zur Vermeidung dieses aufwendigen Nachpflegens von Indexen wird am ursprünglichen Speicherort der betroffenen Datensätze jeweils ein Zeiger (Stellvertreter) auf den neuen Speicherort abgelegt (*Abbildung 47*). Im Zusammenhang mit Satzauslagerungen wird auch von *migrierten Tupeln* gesprochen.

Besonders beim Verschieben langer Datensätze entstehen im ursprünglichen Block Freispeicherlücken, die beim sequenziellen Suchen zu einer Aufwandserhöhung führen. Ein sofortiges Verfolgen der Auslagerungszeiger würde durch die damit verbundenen Sprünge bei sequenziellem Suchen ebenfalls zu einer u.U. erheblichen Aufwandserhöhung führen. Da nach dem relationalen Datenmodell Tupel aber prinzipiell ungeordnet vorliegen, ist ein Verfolgen der Auslagerungszeiger zur Einhaltung bestimmter Reihenfolgen nicht notwendig.

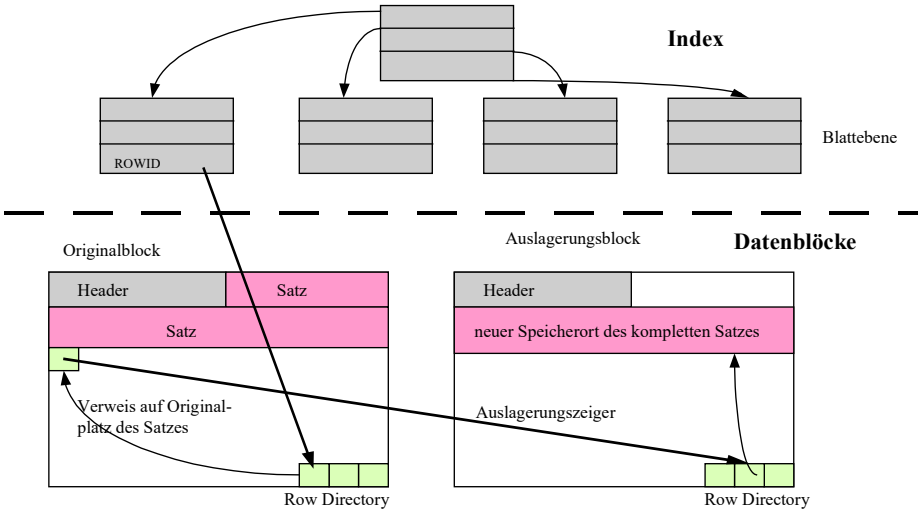


Abbildung 47: Satzauslagerung

Sortierungen werden oft erst vorgenommen, nachdem die Treffermenge einer Suchoperation zusammengestellt wurde. Wird aber über Index Lookups auf solche ausgelagerten Datensätze zugegriffen, so müssen die Zeiger verfolgt werden. Die damit verbundene Erhöhung des Aufwands (also Höhe des Indexbaums plus zwei Zugriffe) deutet der Zugriff Nummer 4 in *Abbildung 48* an.

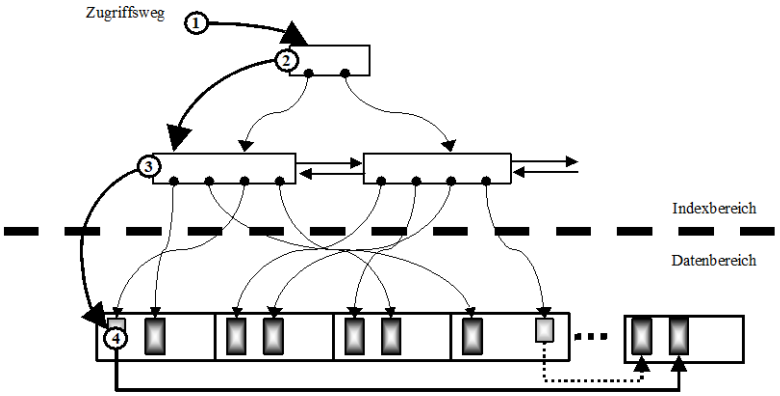


Abbildung 48: Index Lookup mit Auslagerung

Zur Unterstützung von Bereichsanfragen (Index Range Scans) werden die Blätter des Indexbaums in den meisten Implementierungen doppelt miteinander verkettet. Bei der Verarbeitung werden dann die einzelnen Sätze gemäß dem Ordnungskriterium der Tabelle gelesen (*Abbildung 49*). Die nacheinander ausgeführten Zugriffe im zu durchsuchenden Bereich werden durch die Nummerierung angedeutet. Der Zugriff mit der Nummer 4 zeigt wieder eine Aufwandserhöhung, die durch das Verfolgen eines vorhandenen Auslagerungszeigers anfällt.

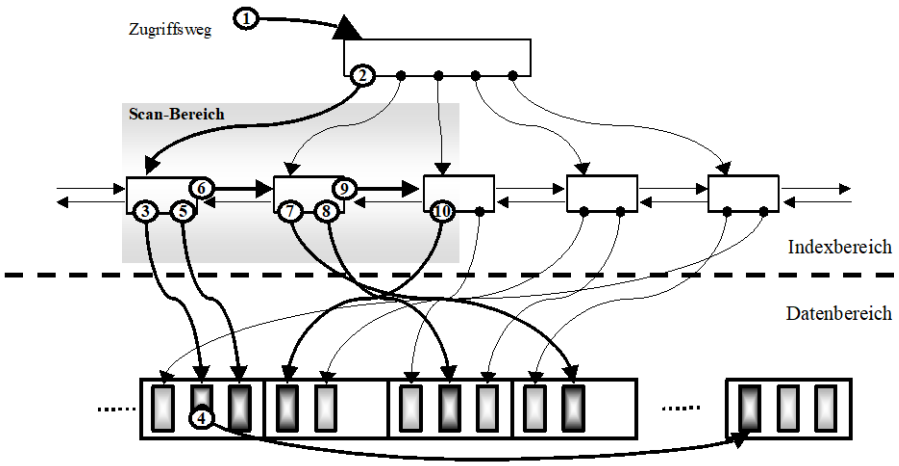


Abbildung 49: Index Range Scan mit Auslagerung

Im vorliegenden Beispiel sind also insgesamt zehn (logische) Blockzugriffe notwendig:

- vier auf Indexblöcke
- fünf auf „normale“ Datenblöcke
- einer auf den Auslagerungsblock

Zur Verringerung der Gefahr von Satzauslagerungen ist es sinnvoll, einen gewissen Speicheranteil in den Datenblöcken für tupelverlängernde Änderungsoperationen freizuhalten. Es ist also nicht immer erstrebenswert, allen Freiplatz in Datenbereichen, z.B. mit einer Reorganisation, zu beseitigen. DBMS-Produkte bieten teils Steuerparameter an (z.B. Oracle mit dem Parameter `PCTFREE`), mit denen festgelegt werden kann, welcher Speicherplatzanteil in Datenblöcken als Reserve für satzverlängernde Änderungsoperationen freigehalten werden soll. Eine weitere Möglichkeit zur Verhinderung der Entstehung von migrierten Tupeln wäre die ausschließliche Verwendung von Datentypen fester Länge bei der Tabellendefinition. Allerdings sind Datentypen variabler Länge, insbesondere auch Zeichenketten variabler Länge, aus Gründen der ökonomischen Verwendung von Speicherplatz eingeführt worden. Würden beispielsweise nur Zeichenketten fester Länge verwendet werden, so würde jeweils Speicher für Zeichenketten der maximalen Länge belegt, unabhängig davon, wie lang die jeweils zu speichernde Zeichenkette wirklich ist. Das führt zu einer Verschwendung von Speicherplatz, die oftmals größere negative Auswirkungen hat als die sukzessive Entstehung von migrierten Tupeln. Diese Alternative kommt also kaum in Frage.

Durch die Speicherung von Daten in Zugangsreihenfolge bei Heap-Tabellen, durch Satzauslagerungen und durch die Möglichkeit, Daten nach unterschiedlichen Kriterien zu indexieren, entspricht die *physische Sortierung* der Daten i.d.R. nicht dem Ordnungskriterium eines jeweils betrachteten Index. Solche Indexe werden auch als *nicht geclusterte Indexe* (Non Clustered Index) [SHS05] bezeichnet. Ein Maß für den Grad der Sortierung von Daten nach dem Ordnungskriterium eines Index ist der sog. *Clustering Factor*. Werden Daten nach einem bestimmten Kriterium sortiert benötigt, so steigt der Sortieraufwand, wenn diese nach dem betrachteten Kriterium

nur wenig vorsortiert gespeichert sind. Auch bei Bereichsanfragen unter Nutzung von Indexen (Index Scans) steigt der Aufwand, weil Datenblöcke evtl. mehrfach gelesen werden müssen.

In der Literatur wird zwischen dünn besetzten und dicht besetzten Indexen unterschieden [SHS05]. Bei *dünn besetzten Indexen* wird nicht für jeden Wert des Indexierungsschlüssels ein Eintrag im Index gespeichert. Liegen die Daten intern nach dem Indexierungsschlüssel sortiert vor, so reicht es aus, wenn im Index ein Eintrag je Datenblock vorhanden ist. Bei *dicht besetzten Indexen* wird für jeden indexierten Datensatz eine entsprechende Zuordnungsinformation gespeichert. Solche Indexe werden üblicherweise verwendet, wenn die Daten ausgelagert (z.B. als Heap organisiert) gespeichert werden. Weiterhin kann noch in eindeutige (unique) Indexe und nicht eindeutige (non unique) Indexe unterschieden werden. Bei einem *eindeutigen Index* kommt jeder Schlüsselwert nur einmal im Index vor und zu jedem Schlüsselwert existiert nur ein Datensatz, auf den verwiesen wird. Eindeutige Indexe werden üblicherweise zur Realisierung von Primärzugriffspfaden genutzt. Bei *nicht eindeutigen Indexen* können die Werte des Indexierungsschlüssels jeweils in mehreren Datensätzen vorkommen. Solche Indexe werden häufig zur Realisierung von Sekundärzugriffspfaden verwendet und daher auch als *Sekundärindexe* bezeichnet. Aus Gründen der Speicherökonomie werden die Schlüsselwerte in den Blättern bei nicht eindeutigen Indexen i.d.R. nur einmal abgelegt. Zu jedem Schlüsselwert wird dann eine Liste mit den Verweisen auf die den Schlüsselwert enthaltenden Datensätze gespeichert (*Abbildung 50*). Würde sich die zu einem Schlüsselwert gehörende Verweisliste über mehrere Blöcke auf der Blattebene erstrecken, so wird in jedem der betroffenen Blätter der Schlüsselwert einmal mit einem jeweils eigenen Teil der Verweisliste gespeichert.

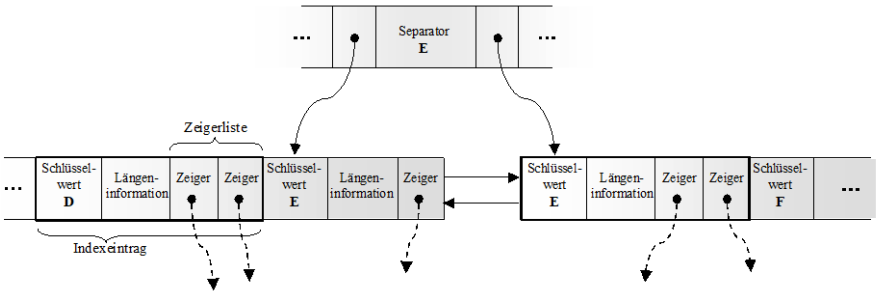


Abbildung 50: Indexeinträge eines Sekundärindex

Insbesondere durch Löschoperationen kommt es auch bei Indexknoten zu einer *Verschlechterung der Speicherausnutzung*. Aus Performance-Gründen werden Verschmelzungen von Indexknoten teilweise von DBMS nicht oder erst sehr spät durchgeführt. Daraus ergibt sich neben der Speicherplatzverschwendung (weniger als 50% des belegten Speichers werden auch tatsächlich genutzt) auch eine Erhöhung der Anzahl der zu lesenden Index-Knoten bei Index Range Scans. Die schlechte Speicherauslastung kann zu einer so stark überhöhten Anzahl Indexknoten führen, dass die *Höhe des Indexbaums* durch eine kompakte Speicherung um eine Ebene verringert werden könnte, was zu einer Reduzierung der für einen Datenzugriff notwendigen Blockzugriffe führt. Bei Einfügeoperationen haben nicht vollständig

gefüllte Indexknoten allerdings wiederum den Vorteil, dass die Wahrscheinlichkeit sinkt, dass ein Aufspalten von Baumknoten notwendig wird. DBMS-Produkte bieten auch hier Steuerparameter an (z.B. Oracle mit dem Parameter `PCTFREE` bzw. `FILLFACTOR` bei Informix), mit denen festgelegt werden kann, welcher Speicherplatzanteil in den Indexblöcken beim Neuaufbau für spätere Einfügeoperationen frei gehalten werden soll. Wenn weiter viele Einfügeoperationen nach dem initialen Aufbau des Index zu erwarten sind, kann dadurch die Systemleistung für einen gewissen Zeitraum verbessert werden. Ist die Speicherplatzreserve erschöpft, muss der Index ggf. neu aufgebaut werden, wenn die Zahl notwendiger Splitting-Operationen auch weiterhin gering gehalten werden soll.

Das physische Löschen von Indexeinträgen erfolgt bei einigen DBMS-Produkten (z.B. DB2, Informix) asynchron. Den Verweisen auf die Datensätze wird meist noch ein Lösch-Flag zugeordnet. Zu löschende Indexeinträge werden über dieses Flag zunächst nur als „gelöscht“ markiert. Später, bspw. in lastarmen Zeiten, wird eine Wartung der Indexstrukturen vorgenommen, bei der u.a. als gelöscht gekennzeichnete Verweise bzw. Indexeinträge (*Ghost-Einträge*) entfernt werden [IBM02]. Ist eine automatisch ablaufende Wartung lange Zeit nicht möglich, z.B. weil der Index stark frequentiert wird, so sammelt sich eine große Anzahl von Ghost-Einträgen an, die den Index ebenfalls aufblähen.

6.5.3 Indexorganisierte Tabellen

Eine Verknüpfung von Daten- und Indexbereichen stellen **indexorganisierte Tabellen** (IOT) dar. Dabei werden die Daten in die Blattebene des Indexbaums integriert. Einfüge-, Änderungs- und Löschoperationen von Daten werden nach den für B^*-Bäume üblichen Methoden vorgenommen. Dadurch sind die Tupel immer nach dem Indexierungsschlüssel sortiert. Deshalb werden solche Strukturen häufig auch als *geclusterte Indexe* (Clustered Index) bezeichnet. Die Sortierung ist insbesondere bei Bereichsanfragen über dem Indexierungsschlüssel vorteilhaft, bzw. wenn Daten häufig nach dem entsprechenden Schlüssel geordnet benötigt werden. Auch hier werden die einzelnen Blätter üblicherweise miteinander doppelt verkettet. Allerdings ist es durchaus wahrscheinlich, dass beim Blocksplitting in der Blattebene der neue Block nicht physisch unmittelbar hinter dem zu teilenden Block reserviert werden kann. Dies führt zu einer Erhöhung des Positionieraufwands der Lese-/Schreibköpfe. Um das Wachstum des Indexbaums zu verlangsamen, ist die Tupellänge entweder systemseitig begrenzt oder das DBMS (bspw. Oracle) bietet die Möglichkeit an, die Sätze zur Speicherung der Tupel aufzuteilen und seltener benötigte Teile in einen *Überlaufbereich* auszulagern (*Abbildung 51*). Durch Löschoperationen kommt es zur langsamen Verschlechterung der Speicherauslastung der Blattknoten. Neben dem höheren Speicherbedarf führt dies bei Bereichsanfragen zu einer Erhöhung der Anzahl zu durchsuchender Blöcke und in extremen Fällen zu einer höheren Ebenenzahl des Index, als eigentlich nötig.

Durch die notwendige Verkettung zwischen dem im Baum gespeicherten Teil eines Tupels und dem ausgelagerten Teil sind, wenn ein gesamtes Tupel gelesen werden soll, weitere Zugriffe für das Lesen des jeweiligen Überlaufblocks nötig. Daher sollte

beim Anlegen von indexorganisierten Tabellen (durch den DBA) darauf geachtet werden, dass die Teile dem Überlaufbereich zugeordnet werden, auf die i.d.R. seltener zugegriffen wird. Die Vorgehensweise ähnelt der vertikalen Partitionierung von Tabellen. Werden bei Änderungsoperationen an den Tupeln Werte der Attribute geändert, die im primären Index, über den die IOT organisiert ist, enthalten sind, so müssen die Sätze physisch verschoben werden, um die Sortierreihenfolge aufrecht zu erhalten.

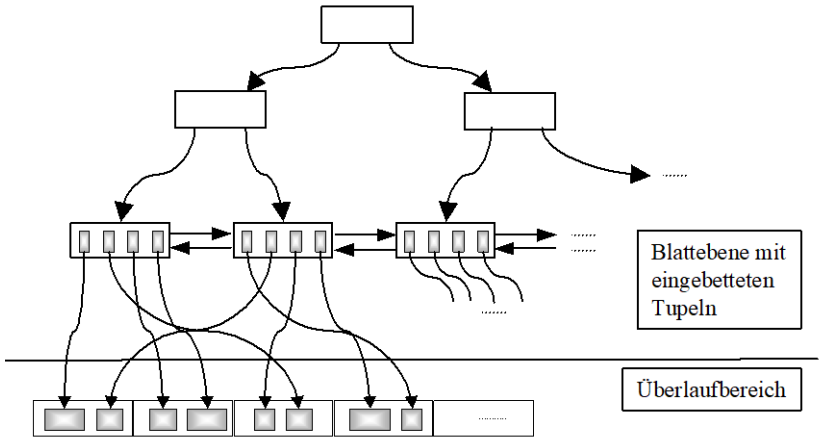


Abbildung 51: Indexorganisierte Tabelle mit Überlaufbereich

Auch beim Teilen von Blattknoten können sich die physischen Speicherorte von Sätzen durch die mit dem Blocksplitting verbundenen Umverteilungen ändern. Bezüglich der bisher betrachteten Adressierung von Datensätzen über physische Referenzen aus zusätzlichen Indexen heraus (TID, RID usw.) hieße dies, dass sich bei solchen Umverteilungen die Tupelidentifikatoren einer größeren Anzahl von Sätzen ändern würden. Das würde entweder zu einem sehr schnellen Anwachsen der Zahl migrierter Tupel oder zu einem erheblichen Aufwand für die Pflege der (anderen betroffenen) Indexe führen. Deshalb werden bei Sekundärindexen auf Tupel indexorganisierter Tabellen häufig logische Referenzen gespeichert (bspw. bei Adabas D oder MaxDB bzw. deren Nachfolgeprodukten). Damit einher geht aber ein erhöhter Aufwand für das Auflösen der logischen Referenzen bei Datenzugriffen. Oracle verwendet hier sog. *logische ROWIDs*, die ihre Gültigkeit auch behalten, wenn ein Satz in der IOT verschoben wird. Eine solche logische ROWID beinhaltet aber auch die Nummer des Blocks, in dem der zugehörige Satz beim Aufbau des Index bzw. bei der letzten Indexwartung gespeichert war [Ora03a]. Damit kann auf Sätze, die seitdem nicht verschoben wurden, mit einem Blockzugriff (zusätzlich zu denen zum Durchsuchen des Sekundärindex) zugegriffen werden. Da im Laufe der Zeit durch die notwendigen Verschiebeoperationen immer mehr dieser Blocknummern ihre Gültigkeit verlieren, muss durch den DBA in gewissen Abständen eine Wartung des Index vorgenommen werden. Diese kann nach [Ora03a] im Online-Betrieb erfolgen.

6.5.4 Unterstützung von Verbundoperationen

Ein Konzept zur Unterstützung von Verbundoperationen ist die tabellenübergreifende Clusterung von Daten (vgl. Abschnitt 6.4.3). Zur Speicherung der Daten eines Clusters wird ein Segment angelegt. Im Unterschied zu Heap-Segmenten und Segmenten zur Datenspeicherung von indexorganisierten Tabellen können hier in einem Bucket (Block) Sätze mit verschiedenem Format (aus unterschiedlichen Tabellen) gespeichert werden. Über den Cluster-Schlüssel wird ein gemeinsamer Zugriffspfad angelegt. Dieser Zugriffspfad kann bei Oracle als B^*-Baum Index oder über eine Hash-Funktion realisiert sein. Im Index wird jeweils einem Wert des Cluster-Schlüssels die Adresse des Bucket zugeordnet, der die den Cluster-Schlüsselwert enthaltenden Sätze aufnimmt. Es werden keine Zuordnungen zu einzelnen Sätzen vorgenommen. Damit stellt der Zugriffspfad eine Ausprägung der dünn besetzten Indexe dar. Oracle verwendet kein erweiterbares Hashing. Wird der Zugriffspfad über eine Hash-Funktion realisiert, so muss beim Anlegen des Clusters angegeben werden, wie viele Buckets sofort im primären Bereich des Clusters angelegt werden sollen. Da Hash-Funktionen i.d.R. nicht eineindeutig sind, können in einem Bucket durchaus Sätze mit unterschiedlichen Cluster-Schlüsselwerten untergebracht werden. Deshalb wird der jeweilige Wert des Cluster-Schlüssels bei Hash-Clustern in den Datensätzen mit gespeichert.

Ist der Zugriffspfad als B^*-Baum Index organisiert, so wird jeweils beim Einfügen eines neuen Cluster-Schlüsselwerts ein neuer Bucket angelegt. Da alle in einem Bucket gespeicherten Datensätze den gleichen Cluster-Schlüsselwert enthalten, wird dieser, um Speicher zu sparen, im Bucket nur einmal abgelegt [Ora03b]. Wächst die Zahl der zu einem Cluster-Schlüsselwert gehörenden Tupel in Laufe der Zeit (stark) an, so reicht u.U. ein Datenblock zu deren Speicherung nicht mehr aus. In diesem Fall werden *Überlaufbereiche* angelegt. Das Verfahren ähnelt im Wesentlichen dem von Hash-Tabellen her bekannten *Separate Chaining*, bei dem je Bucket eine separate Überlaufkette angelegt wird. Durch das Durchsuchen der Überlaufketten steigt der Aufwand für Datenzugriffe (*Abbildung 52*), der im Idealfall, neben den für das Durchsuchen des Index notwendigen Zugriffen, nur einen Zugriff auf Datenblöcke betragen sollte. Überlaufbereiche können durch eine Reorganisation beseitigt bzw. verkleinert werden, wenn diese durch das Löschen von Tupeln nur noch teilweise gefüllt sind oder wenn bei einer Reorganisation die Bucket-Größe erhöht werden kann.

Zusätzlich zum Cluster-Index können noch weitere Indexe (zusätzliche Indexe) erzeugt werden, die sich auf einzelne im Cluster gespeicherte Tabellen beziehen. Bei der Indexierung wird hier wie bei Heap-Tabellen vorgegangen. In den Blättern der Indexbäume werden physische Verweise auf die Datensätze gespeichert. Durch satzverlängernde Änderungsoperationen kann es auch bei Clustern zum Verschieben der Sätze in andere Datenblöcke, der dem Bucket zugeordneten Überlaufkette kommen.

Dies führt bei Zugriffen über zusätzliche Indexe, wie bei Heap-Tabellen auch, neben der Entstehung von eingestreutem Freiplatz, zu einer Erhöhung des Suchaufwands durch die Verfolgung der Auslagerungszeiger. Ändern sich bei Tupeln die Werte von

Attributen, über denen der Cluster-Schlüssel definiert ist, so müssen die zugehörigen Sätze physisch verschoben werden, damit die Cluster-Eigenschaft erhalten bleibt. Um auch hier das Nachpflegen eventuell vorhandener zusätzlicher Indexe zu vermeiden, wird ebenfalls mit Auslagerungszeigern gearbeitet, die am ursprünglichen Speicherort abgelegt werden.

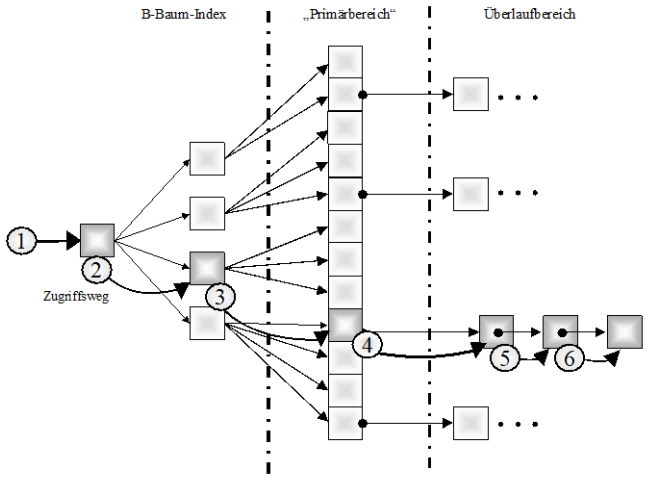


Abbildung 52: Datenzugriff bei einem Index-Cluster

Bei Zugriffen über den Cluster-Schlüssel existiert kein direkter Zusammenhang zwischen der Zahl der migrierten Tupel und dem Suchaufwand. Eine Erhöhung des Suchaufwands ergibt sich aber indirekt durch die mit den Auslagerungen (durch die Entstehung eingestreuten Freiplatzes) oftmals verbundene Verlängerung der jeweiligen Überlaufketten. Diese Verlängerung der Überlaufketten führt zu einer Erhöhung der Anzahl Datenblöcke, die der Cluster insgesamt belegt, und damit auch zu einer Erhöhung des Aufwands für sequenzielles Durchsuchen einzelner Tabellen des Clusters. Die Verwendung von Clustern ist aus praktischer Sicht zu empfehlen wenn:

- die Cluster-Schlüsselwerte der einzelnen Tupel nur selten geändert werden,
- die Daten der im Cluster gespeicherten Tabellen oft gemeinsam (über den Cluster-Schlüssel verbunden) benötigt werden,
- die im Cluster gespeicherten Tabellen wenig wachsen,
- die Cluster-Gruppen (siehe Abschnitt 6.4.3) möglichst annähernd gleiche Größen haben und
- die einzelnen Tabellen des Clusters selten sequenziell durchsucht werden.

Eine *verallgemeinerte Zugriffspfadstruktur* (Generalized Access Path) zur Unterstützung von Verbundoperationen zwischen hierarchisch in Beziehung stehenden Datenbankobjekten bzw. zur Überprüfung referenzieller Integritäten, die ebenfalls auf B*-Baum-Variationen basiert, wird in [HR01] vorgeschlagen. Üblicherweise werden auf der Blattebene eines Index nur Verweise auf Datensätze *einer* Tabelle gespeichert. Das heißt, ein Index wird genau einer Tabelle zugeordnet. Bei der verallgemeinerten Zugriffspfadstruktur wird ein Index mehreren,

typischerweise in hierarchischer Beziehung zueinander stehenden Tabellen zugeordnet. Einem Schlüsselwert im Index wird jeweils ein Verweis auf den Datensatz des in der Hierarchie übergeordneten Objekts und eine Liste mit Verweisen auf die Datensätze untergeordneter Objekte gespeichert. *Abbildung 53* zeigt ein Beispiel, für die Verknüpfung von Mitarbeiterdaten mit den Daten der Abteilungen, denen sie zugeordnet sind.

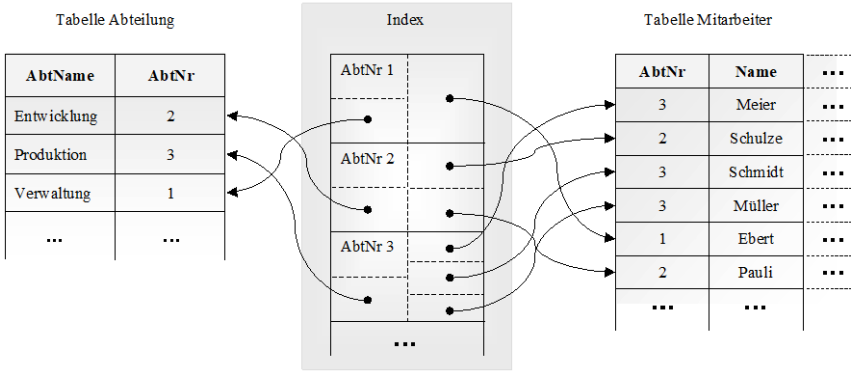


Abbildung 53: Verallgemeinerte Zugriffspfadstruktur zur Verknüpfung von Mitarbeiter- und Abteilungsdaten

Der Index ist hier nur schematisch dargestellt. Er kann beispielsweise als B^*-Baum realisiert und als Primärzugriffspfad für den Zugriff auf die Sätze der übergeordneten Objekte, als Sekundärzugriffspfad für den Zugriff auf die Daten der in der Hierarchie untergeordneten Objekte und als gemeinsamer Zugriffspfad (bspw. zur Unterstützung von Verbundoperationen) verwendet werden. Zu beachten ist, dass der Generalized Access Path wegen seiner Größe natürlich nicht immer optimal ist (bspw. beim Primärzugriff gegenüber einem reinen Primärindex). Der Index kann unabhängig von der Organisation der Datenbereiche und von anderen Indexen aufgebaut werden. Eine bestimmte physische Anordnung der Daten wird nicht erzwungen und muss somit auch nicht, wie beim oben betrachteten Cluster, gepflegt werden. Die Daten der über- und der untergeordneten Tabellen können, wie bei relationalen DBMS üblich, in physisch getrennten Bereichen gespeichert werden. Verbundoperationen werden, wie beim Cluster auch, durch den gemeinsamen Zugriffspfad auf die Daten der am Verbund beteiligten Tabellen unterstützt.

Der Aufwand für die reinen Datenzugriffe ist durch deren physisch getrennte Speicherung allerdings i.d.R. höher als beim Cluster. Durch weitere Verallgemeinerungen kann das Einsatzgebiet noch erweitert werden. So sind mehrere Verweislisten auf untergeordnete Objekte verschiedenen Typs, die in verschiedenen Tabellen gespeichert sind, denkbar. Werden für alle Tabellen Verweislisten verwendet, so können auch Verbundoperationen über in allen Tabellen enthaltene Schlüssel unterstützt werden.

6.6 Unterstützung komplexer Objekte

In einigen Anwendungsbereichen (z.B. im wissenschaftlich-technischen Bereich) mit großen Mengen komplex strukturierter Daten führt die Beschränkung auf Relationen

mit atomaren Attributen oftmals dazu, dass logisch eng verknüpfte Daten, wie z.B. die eines komplexen Objekts, bei rein relationalen DBMS auf mehrere Tabellen verteilt werden müssen. Das führt u.U. zu einem erheblichen Aufwand bei der Rekonstruktion der Objekte. Um Anwendungen mit komplex strukturierten Daten besser unterstützen zu können, sind die Anbieter relationaler DBMS teilweise (bspw. Oracle bzw. IBM mit DB2 und Informix) dazu übergegangen, ihre Systeme um objektorientierte Konzepte zu erweitern und damit (zumindest ansatzweise) objektrelationale Datenbank-Management-Systeme (ORDBMS) anzubieten. Diese Systeme lassen die Modellierung komplexer Objekte ohne strenge Einhaltung der ersten Normalform für Relationen zu. Damit wird zwar zunächst das im Zusammenhang mit relationalen Systemen und Anwendungserfordernissen oftmals genannte Problem des *impedance mismatch* abgemildert, das Problem der aufwendigen Rekonstruktion komplexer Objekte aus den in flacher relationaler Form gespeicherten Daten bleibt jedoch bestehen. Es ist wünschenswert, auch die Abbildung komplexer Objekte auf die vorhandenen physischen Speicherungsstrukturen der logischen Strukturierung der Objekte entsprechend gestalten zu können.

6.6.1 Abbildungsmöglichkeiten auf physische Speicherungsstrukturen

Objektrelationale Datenbank-Management-Systeme müssen die Verwaltung relational repräsentierter Daten in flachen Tabellen und die Speicherung und Verarbeitung komplexer und geschachtelter Datenobjekte parallel ermöglichen. Eine zentrale Anforderung ist dabei, dass die Integration neuer objektrelationaler Konstrukte und die neu gewonnene Erweiterbarkeit des DBMS die Performance bei der Verarbeitung der bisherigen flachen Tabellen nicht beeinträchtigt. Durch die Beschränkung der Erweiterungen und Änderungen auf die oberen Architekturschichten Datensystem und Zugriffssystem und auf die Anfrageübersetzung ist dieses Ziel erreichbar [HR01]. Die Komponenten zur Puffer-, Block- und physischen Satzverwaltung können weitgehend unverändert bleiben.

In verschiedenen Forschungsarbeiten (z.B. [Keß95, Ska06]) werden verschiedene Abbildungsmöglichkeiten auf interne Satzstrukturen und Clusterungs-Strategien für komplexe Objekte untersucht und beschrieben. Darüber hinaus werden dort Sprachkonstrukte definiert, die Anwendern Einflussmöglichkeiten auf die physische Speicherungsstruktur geben.

Dabei werden als Freiheitsgrade

- die Aufteilung von Objekten auf mehrere physische Sätze,
- seitenüberspannende Sätze,
- der Speicherort physischer Sätze,
- Typen von Objektreferenzen sowie
- die physische Kollektionsstruktur und
- Clusterungs-Strategien

genannt.

Unterschiedliche Möglichkeiten der Abbildung von komplexen Objekten sollen am vereinfachten Beispiel eines Telekommunikationsdienstes dargestellt werden. Der Telekommunikationsdienst „Universal Number“ bietet sog. Dienstkunden (z.B. Pannendienste, Versicherungen etc.) die Möglichkeit, unter einer landesweit einheitlichen („virtuellen“) Rufnummer (z.B. 0180-189765) erreichbar zu sein. Die einzelnen Anrufe sollen aber nicht in einem zentralen Call-Center bearbeitet werden, sondern von den Vermittlungsanlagen direkt an die den Nutzern der Dienste jeweils am nächsten liegende Niederlassung des Dienstkunden weitergeleitet werden. Hauptaufgabe des sog. Verkehrsführungsprogramms, das den Dienst realisiert, ist die Umsetzung der einheitlichen virtuellen Rufnummer in eine zugehörige reale Rufnummer. Eine entsprechende Modellierung der Daten für diesen Dienst zeigt *Abbildung 54* in Form eines Objektdiagramms.

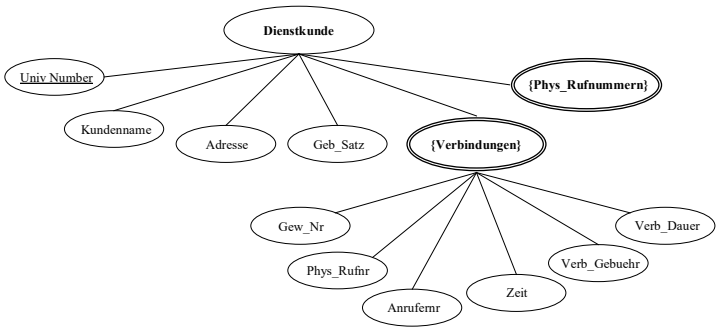


Abbildung 54: Objektdiagramm des Beispiels

Neben den üblichen Daten wie Kundenname, Adresse und Gebührensatz werden unseren Annahmen zufolge im Objekt `Dienstkunde` im Kollektionsattribut `Verbindungen` noch die Daten aller über das Telekommunikationsunternehmen abgewickelten Verbindungen gespeichert. Über die Telefonnummer des Anrufers (`Anrufernr`) und die Anrufzeit werden die einzelnen Verbindungen identifiziert. Weiterhin werden die vom Anrufer gewählte Nummer, die reale („physische“) Rufnummer, zu der der Anruf weitergeleitet wurde, die Verbindungsdauer sowie der tatsächlich angefallene Gebührensatz gespeichert. Im mengenwertigen Attribut `Phys_Rufnummern` werden den jeweiligen virtuellen die realen Rufnummern zugeordnet. Das auf der Datenbank arbeitende Verkehrsführungsprogramm für den Dienst „Universal Number“ bestimmt anhand der gewählten Nummer (z.B. 0180-189765) den entsprechenden Dienstkunden und eine passende verfügbare reale Rufnummer, mit der der Anrufer verbunden wird. Nach dem Anruf wird dieser im Kollektionsattribut `Verbindungen` verzeichnet.

Für eine zunächst rein relationale Umsetzung des Beispiels erfolgt die Darstellung in einem E/R-Diagramm ohne Nutzung erweiterter Konzepte, wie mengenwertige und strukturierte Attribute (*Abbildung 55*).

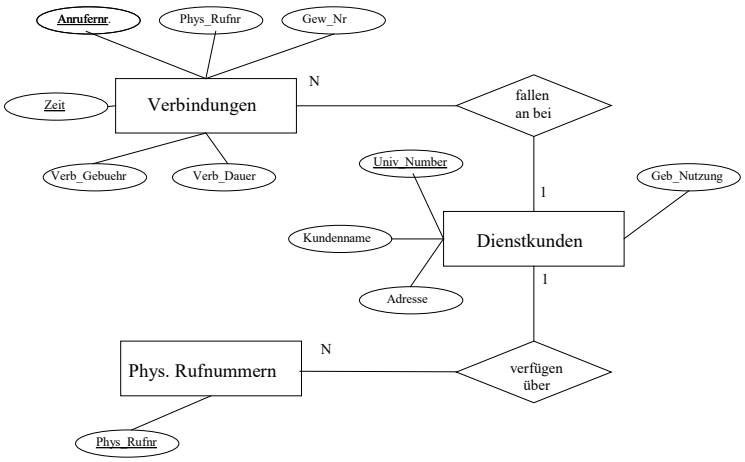


Abbildung 55: Informationsschema des Beispiels

Nach den üblichen Transformationsregeln ergibt sich daraus etwa das in *Abbildung 56* dargestellte relationale Schema. Primärschlüsselattribute sind unterstrichen und Fremdschlüsselattribute kursiv dargestellt. Das Attribut `Gew_Nr` enthält die vom Anrufer gewählte Nummer, die der `Univ_Number` des Dienstkunden entspricht.

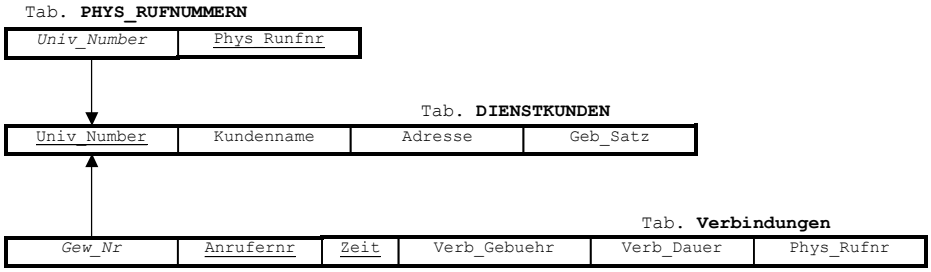


Abbildung 56: Relationales Schema für das Beispiel „Universal Number“

Die Abbildung der drei Tabellen kann auf die in *Abschnitt 6.5* beschriebenen Strukturen erfolgen.

Bei der Speicherung der Daten dieses Umweltausschnitts als komplexes Objekt in einer objektrelationalen Datenbank sind hingegen verschiedene physische Repräsentationen möglich.

Die Daten komplexer Objekte werden prinzipiell in Satzstrukturen gespeichert, die im Unterschied zu rein relationalen Implementierungen allerdings beliebig geschachtelt werden können. Die einzelnen Komponenten eines Satztyps, die selbst wieder Sätze sein können, werden entweder in die Satzstruktur *eingebettet* oder aus ihr heraus *referenziert* (*Abbildung 57*). Bei einer Referenzierung von Komponenten werden deren Daten physisch entfernt (ausgelagert) von der Satzstruktur des umfassenden Objekts (Hauptobjekt) gespeichert, die lediglich einen Verweis auf den Speicherort der Daten der jeweiligen Komponente (Subobjekt) enthält.

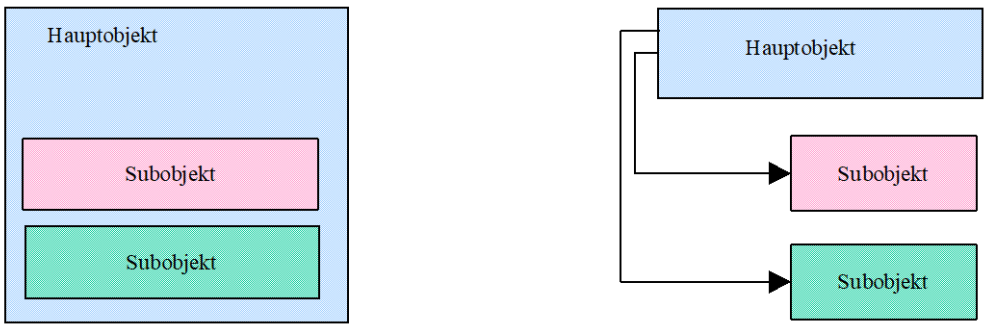


Abbildung 57: Eingebettete und referenzierte Speicherung komplexer Objekte

Im betrachteten Beispiel bietet sich besonders eine Auslagerung der Verbindungsdaten an, da die Anzahl anfallender Verbindungen jeweils sehr unterschiedlich und schwer abschätzbar ist und hier mit einer starken Zunahme des Umfangs der zu speichernden Daten gerechnet werden muss. Die Entscheidung, ob die Menge der realen Rufnummern, in die die Daten der jeweiligen Dienstkunden enthaltenden Sätze eingebettet gespeichert werden kann oder nicht, ist u.a. davon abhängig, wie stark die Anzahl der Rufnummern schwankt und ob eine Obergrenze existiert. Ob physische Sätze auf die Größe eines Datenblocks beschränkt sind oder ob diese mehrere Blöcke *überspannen* dürfen muss hier evtl. ebenfalls berücksichtigt werden.

Neben der Festlegung der physischen Satzstruktur kann für die einzelnen Satztypen komplexer Objekte über die Angabe des jeweiligen Table Space der physische *Speicherort* auf den Sekundärspeichermedien grob festgelegt werden.

Referenzen auf Objekte können die physische Adresse des Speicherorts enthalten oder über logische Werte realisiert werden. Bei der Verwendung logischer Werte müssen diese mit Hilfe einer Indexstruktur in physische Adressen umgesetzt werden. Auch Mischformen, bestehend aus logischen Werten mit Hinweisen auf physische Speicherorte, sind möglich.

Mehrere Sätze eines Typs können fortlaufend in Heap-, verketteten Listen-, Array- oder Baum-Strukturen abgelegt werden. Bei verketteten Listenstrukturen werden die Speicherbereiche, die die einzelnen Sätze belegen, durch Zeiger miteinander verbunden. Dies ermöglicht eine effiziente Zusammenfassung auch dann, wenn die einzelnen Sätze unterschiedliche Größen haben können oder wenn sich die Größe von Sätzen durch Update-Operationen u.U. erheblich ändern kann. Zur einfachen Anwendung der bei Arrays i.d.R. üblichen Methode der indexierten Adressierung muss sichergestellt werden, dass alle Sätze die gleiche Größe besitzen. Die einzelnen Sätze werden dann nacheinander in den einzelnen Elementen des Array abgelegt. Um den Speicherplatz für das Array vorab reservieren zu können und damit eine eingebettete Speicherung zu ermöglichen, muss auch die (maximale) Anzahl der Elemente bekannt sein. Durch die Verwendung der beschriebenen Strukturen als Satzkomponenten können verschiedene Arten von *Kollektionen* abgebildet werden. Die eingebettete Speicherung der Menge der realen Rufnummern könnte bspw. über ein Array erfolgen.

6.6.2 Clusterung der Daten komplexer Objekte

Zwei wichtige *Strategien zur Clusterung* von Sätzen sind die objektbezogene Cluster-Bildung und die objektübergreifende Cluster-Bildung. Bei der objektübergreifenden Cluster-Bildung werden physische Datensätze des gleichen Typs (der gleichen Tabelle) möglichst dicht gespeichert. Diese Speicherungsform entspricht dabei im Wesentlichen der bei relationalen DBMS üblichen Speicherungsform. In Tabellen gespeicherte Fremdschlüsselwerte können dabei als eine Art logischer Referenzen angesehen werden. *Abbildung 58* zeigt diese Form der Cluster-Bildung bezogen auf das Beispiel des Telekommunikationsdienstes. Die einzelnen Cluster sind dabei jeweils grau hinterlegt.

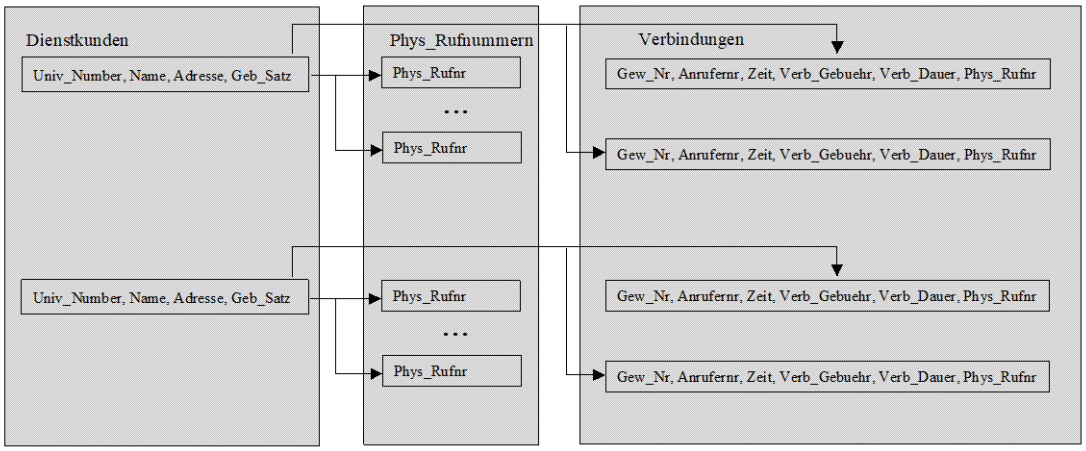


Abbildung 58: Objektübergreifende Cluster-Bildung

Müssen bspw. die Daten aller Dienstkunden oftmals gemeinsam, unabhängig von den realen Rufnummern, verarbeitet werden, so bewirkt die Auslagerung der realen Rufnummern eine Verringerung der zu verarbeitenden Datenmenge. Die physisch beieinanderliegende Speicherung der Sätze aller Dienstkunden führt weiterhin zu einer Verkleinerung der Positionierungsoperationen der Lese-/Schreibköpfe beim sequenziellen Suchen. Unterscheidet sich die Zahl der realen Rufnummern bei den verschiedenen Dienstkunden stark und ist deren maximale Zahl je Dienstkunde ebenfalls nicht absehbar, so ist die von den übrigen Daten der Dienstkunden getrennte Speicherung auch einfacher zu realisieren, als die eingebettete Speicherung der realen Rufnummern.

Werden die Daten der Dienstkunden und die zugehörigen realen Rufnummern allerdings zusammen benötigt, wie dies bspw. bei der Vermittlung von Anrufen der Fall ist, so verursacht die physische getrennte Speicherung Aufwand für die Verknüpfung der Daten. Hier bietet sich die objektbezogene Cluster-Bildung als wesentliche Möglichkeit objektrelationaler DBMS an (komplexe Objekte auf Modell- und *Speicherungsebene*). Dabei werden die physischen Sätze, die zu einem komplexen Objekt gehören, dicht gespeichert, obwohl unterschiedliche Satztypen vorliegen. Alle Sätze von Subobjekten (auch Mitgliedssätze oder sekundäre Sätze) werden in unmittelbarer Nähe des Satzes des jeweils umfassenden Objekts (auch

Primärsatz) gespeichert. *Abbildung 59* zeigt die objektbezogene Cluster-Bildung am Beispiel des Telekommunikationsdienstes.

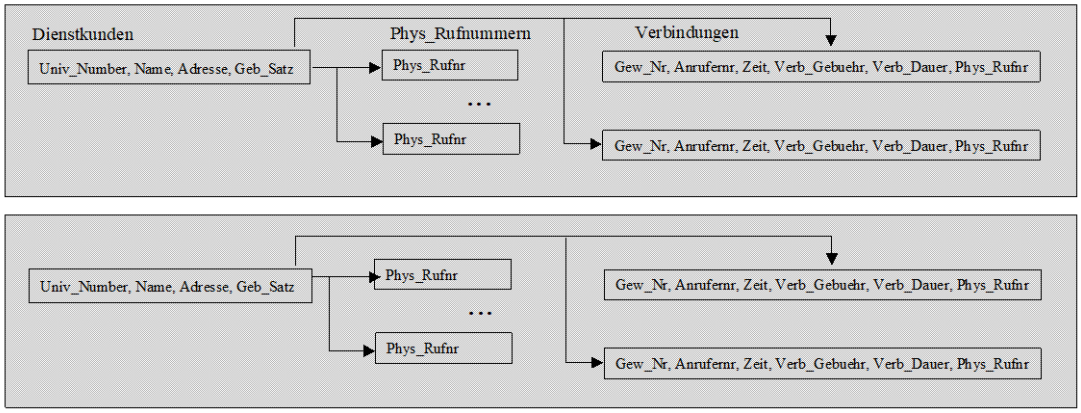


Abbildung 59: Objektbezogene Cluster-Bildung

Mit der von Oracle implementierten Cluster-Struktur kann diese Form der Cluster-Bildung realisiert werden.

Die verallgemeinerte Zugriffspfadstruktur und die Cluster-Struktur von Oracle (vgl. Abschnitte 6.4.3 und 6.5.4) unterstützen eine hierarchische Clusterung von Datenbankobjekten auf zwei Ebenen. In [ZSL98] wird unter dem Begriff *Dynamic Hierarchical Clustering* eine Methode vorgeschlagen, die eine Clusterung der Sätze hierarchisch strukturierter Datenobjekte ermöglicht, die aus mehr als zwei Ebenen bestehen. Als Speicherungsstruktur werden B^*-Baum-Variationen verwendet, bei denen die Daten komplexer Objekte, inklusive der Daten von Subobjekten, auf der Blattebene gespeichert werden. Dadurch bleibt die physisch beieinanderliegende Speicherung der Daten komplexer Objekte auch bei der Anwendung von Lösch- und Einfügeoperationen erhalten. Um die Clusterung zu erreichen, werden zusammengesetzte (Pfad)Schlüssel gebildet, über denen der Indexbaum aufgebaut wird. Beispielsweise kann für Mitarbeiter, die in den Niederlassungen von Unternehmen beschäftigt sind, ein Schlüssel (*Look Up Key*) aus der Kombination `Unternehmen/Niederlassung/Mitarbeiter` gebildet werden. Zur Reduzierung der Größe der Schlüsselwerte wird eine als *„Enc“* bezeichnete numerische Codierung vorgenommen. Auf der obersten Ebene werden die Objekte (bspw. die Unternehmen) in der Reihenfolge ihres Einfügens nummeriert. Die zu einem Objekt gehörenden Subobjekte (hier bspw. die Niederlassungen) werden wiederum in der Einfügereihenfolge nummeriert. Diese Nummern werden als C*hild Number* bezeichnet. Die Nummerierung wird bis zur unteren Ebene (hier Mitarbeiter) fortgesetzt. Würden beispielsweise die in *Tabelle 4* aufgeführten Daten von Firmen, zugehörigen Niederlassungen und deren Mitarbeitern eingefügt, so würden sie wie ebenfalls in Tabelle 4 zusätzlich angegeben codiert und wie in *Abbildung 60* dargestellt abgespeichert werden. Die in Abbildung 60 unten rechts angedeutete Slot-Liste spiegelt auch die Einfügereihenfolge wieder. Dabei wird angenommen, dass die Slot-Liste vom Ende her in den Block „hineinwächst“.

Firma (Code)	Niederlassung (Code)	Mitarbeiter (Code)
Meier AG (1)	Berlin (1-1)	Müller (1-1-1)
		Schulze (1-1-2)
	München (1-2)	Franz (1-2-1)
Schmidt GmbH (2)	Jena (2-1)	

Tabelle 4: Beispieldaten für Dynamic Hierarchical Clustering

Den Datensätzen der Objekte werden intern noch die Look Up Keys hinzugefügt. Zusätzlich wird bei den Objekten, die Subobjekte besitzen können, noch der nächste zu vergebende Child Key (in *Abbildung 60* in Klammern dargestellt) gespeichert. Bei Einfügeoperationen erfolgt eine Top-Down-Suche. Wurde das dem einzufügenden Objekt übergeordnete Objekt gefunden, wird aus dessen Look Up Key und dem nächsten zu vergebenden Child Key der Look Up Key des einzufügenden Objekts gebildet und der Datensatz für das Objekt eingefügt.

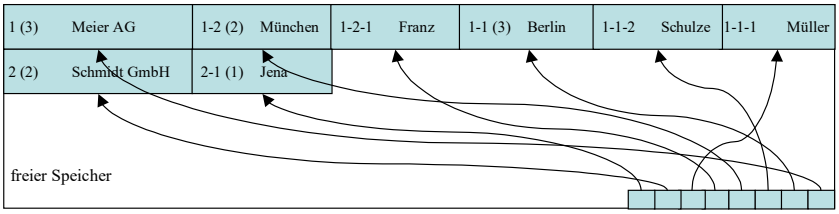


Abbildung 60: Datenspeicherung beim Dynamic Hierarchical Clustering

Der Speicherbedarf eines Objekts und seiner Subobjekte kann den in einem Block auf der Blattebene verfügbaren Speicher übersteigen. Bei den für B-Bäume üblichen Einfügeverfahren würden die Daten neuer Subobjekte nach einer Splitting-Operation auf der Blattebene aber nicht mehr zwingend in den Block eingefügt werden, der auch die Daten des übergeordneten Objekts enthält. Dies würde dazu führen, dass beim Einfügen neben dem Block, der das übergeordnete Objekt enthält, noch ein weiterer Block, in den das Subobjekt eingefügt wird, geändert werden muss. Die Änderung des das übergeordnete Objekt enthaltenden Blocks ist zur Änderung des nächsten zu vergebenden Child Key notwendig. Durch eine leichte Modifikation der Vergleichsoperation beim Einfügen wird dafür gesorgt, dass neue Subobjekte immer in den Block eingefügt werden, der auch die Daten des übergeordneten Objekts enthält. Somit muss bei Einfügeoperationen, außer wenn aktuell ein Block-Splitting nötig wird, nur ein Block geändert werden.

Verallgemeinerte Zugriffspfade sowie die eben beschriebenen Speicherungsstruktur zur Cluster-Bildung werden nach dem derzeitigen Kenntnisstand von am Markt verfügbaren und im kommerziellen Einsatz verbreiteten Systemen nicht zur Verfügung gestellt. Allerdings könnten die vorgeschlagenen Konzepte durch relativ geringe Änderungen an vorhandenen Strukturen im Rahmen der Weiterentwicklung von (objektrelationalen) DBMS-Produkten umgesetzt werden. Weitergehende Probleme, etwa die adäquate Berücksichtigung bei der Anfrageoptimierung, Synchronisationsverfahren etc. seien hierbei in der Bewertung erst einmal außer Acht gelassen.

6.7 Mehrdimensionale Zugriffspfade – Grid File

An mehrdimensionale Zugriffspfade, wie sie bspw. bei Geoinformationssystemen benötigt werden, werden andere Anforderungen gestellt. Suchprobleme beziehen sich hier meist nicht auf normale Vergleiche (<,>,=,...), sondern es werden Punkt- oder Gebietsanfragen gestellt (z.B. Finden des nächsten Nachbarn). Ein Problem bei der Indexierung stellt die i.d.R. variierende Objektdichte in verschiedenen Bereichen dar (*Abbildung 61*).

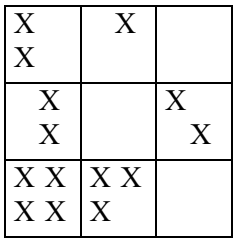


Abbildung 61: Bereiche mit variierender Objektdichte

Da die Größe der Blöcke zum Speichern von Daten bei DBMS i.d.R. nicht dynamisch variiert werden kann, müssen die Daten stark belasteter Bereiche auf mehrere Blöcke verteilt werden. In der Literatur existieren unterschiedliche Vorschläge für die Realisierung mehrdimensionaler Zugriffspfade (Quadranten-Bäume, U-B-Bäume, mehrdimensionales Hashing usw.) . Hier soll beispielhaft eine Lösung, das sog. Grid-File, kurz betrachtet werden. Als Ausgangspunkt dient der *Abbildung 62* in dargestellte Datenraum. Die Kreuze könnten z.B. Lagerorte von Bodenschätzen darstellen, die in verschiedenen Quadranten einer Landkarte liegen.

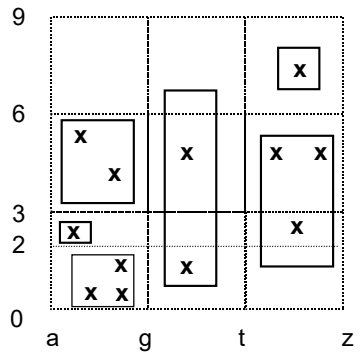


Abbildung 62: Datenraum für Beispiel

Zur Speicherung der Daten der Lagerorte werden Buckets verwendet, die jeweils die Daten von bis zu drei Orten aufnehmen können. Die Rechtecke, die um die Kreuze gezogen sind, fassen die Lagerorte zusammen, deren Daten im gleichen Bucket gespeichert werden. Die Buckets sollen mit A bis F bezeichnet werden. Der Index des Grid-File besteht aus mehreren verketteten Listen (eine je Dimension). Die Listen enthalten in jedem Knoten einen entsprechenden Achsenwert (je nach Dimension) sowie einen Verweis auf eine Zeile bzw. Spalte des sog. GD-Bereichs. Dieser Bereich enthält dann die Verweise auf die Buckets, in denen die eigentlichen Daten gespeichert sind. Die Anzahl Zeilen und Spalten des GD-Bereichs ist ebenfalls

variabel. Zeilen oder Spalten können bei Bedarf beim Einfügen neuer Daten (im Beispiel über die Orte neu entdeckter Lagerstätten) geteilt werden. In *Abbildung 63* wurde die Zeile 3 als letzte zum GD-Bereich hinzugefügt, als der Bereich 0 bis 3 der vertikalen Dimension aufgeteilt werden musste.

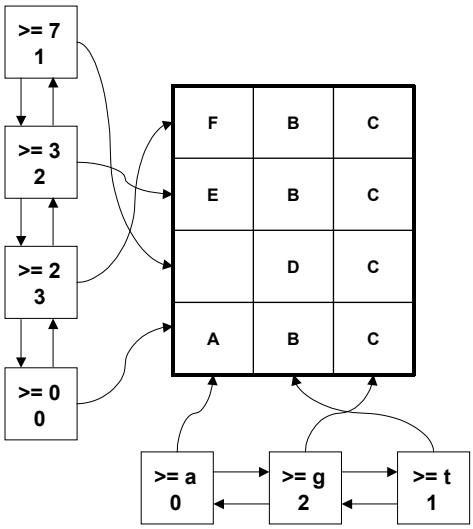


Abbildung 63: Grid-File-Struktur für das Beispiel

7 Anfrageverarbeitung

7.1 Anfrageübersetzung

SQL-Anweisungen beschreiben Eigenschaften, die bspw. ein Suchergebnis erfüllen soll. Sie enthalten keine Elemente, die angeben, wie die Anweisungen ausgeführt werden sollen. Bei der DBMS-internen Abarbeitung von Anweisungen werden mittels relativ einfacher Grundoperationen (*Planoperatoren*) die notwendigen I/O-Operationen ausgeführt. Es ist u.a. die Aufgabe des Anfrageoptimierers, Folgen von Planoperatoren (*Ausführungspläne*) zu finden, mit denen gegebene SQL-Anweisungen *in der konkreten Systemumgebung* auf kostengünstige Weise realisiert werden können.

Der gesamte Prozess der Übersetzung von Anfragen erfolgt in den folgenden Schritten.

- lexikalische und syntaktische Analyse
- semantische Analyse (Existieren die gewünschten Objekte?)
- Zugriffs- und Integritätskontrolle
- Standardisierung und Vereinfachung (z.B. Bedingungen in WHERE-Klauseln in disjunktive oder konjunktive Normalform bringen)
- Restrukturierung und Transformation (Erstellung von Ausführungsplänen, Anfrageoptimierung)
- Code-Generierung für eine direkte oder interpretative Ausführung

7.2 Anfrageoptimierung

Wenn eine SQL-Anweisung gültig ist (syntaktisch, semantisch, ...) und ausgeführt werden soll, muss ein möglichst kostengünstigster Weg gefunden werden, die Anweisung auszuführen. Dabei soll bspw. auch die *Reihenfolge der Bedingungsausdrücke* in SQL-Anweisungen *keine Auswirkung* auf das Zugriffsverhalten haben. Es ist die Aufgabe des Anfrageoptimierers, diesen Weg zu finden.

Zur Ausführung einer Anweisung erstellt der Optimierer zunächst einen oder mehrere Ausführungspläne. Anfrageoptimierer (z.B. auch der von Oracle) können regelbasiert oder kostenbasiert arbeiten. Bei der regelbasierten Optimierung arbeitet der Optimierer nach den Regeln der Relationenalgebra (z.B. Selektion vor Join etc.). Der tatsächliche Zustand (z.B. Datenmengen, Datenverteilung) der Datenbank wird dabei nicht berücksichtigt. In bestimmten Fällen (z.B. bei sehr kleinen aber indexierten Tabellen) liefert die regelbasierte Optimierung ungünstige Zugriffspfade. Abhilfe schafft hier die kostenbasierte Optimierung, bei der mit Hilfe von Statistiken der tatsächliche Datenbankzustand berücksichtigt wird. Mit Hilfe dieser Statistikdaten werden die Ausführungskosten für die verschiedenen Ausführungspläne geschätzt und es wird der ausgewählt, der die niedrigsten Ausführungskosten erwarten lässt.

Voraussetzung für eine kostenbasierte Optimierung sind aktuelle Statistikdaten. Diese Daten werden im Datenbankkatalog gespeichert. Wegen des i.d.R. hohen Aufwands werden die Statistikdaten von DBMS im laufenden Betrieb nicht oder nicht vollständig gepflegt. Daher ist es für einen DBA ratsam, die Statistikdaten regelmäßig (möglichst zu lastarmen Zeiten) und nach größeren Änderungen am Datenbestand (z.B. nach der Archivierung größerer Datenmengen) zu aktualisieren. Die Aktualisierung der Statistiken erfolgt bei Oracle mit der Anweisung

```
ANALYZE {TABLE | INDEX} <name> COMPUTE STATISTICS;
```

Dies ist die einfachste Form der Anweisung (weitere Informationen siehe Dokumentation).

Beispiele:

```
ANALYZE TABLE abteilung COMPUTE STATISTICS;

ANALYZE INDEX i_anr COMPUTE STATISTICS;
```

Zu Analysezwecken kann sich der Datenbankadministrator die vom Anfrageoptimierer ermittelten Ausführungspläne anzeigen lassen. Aus diesen Anzeigen können z.B. Schlüsse gezogen werden, ob eine Aktualisierung der Statistikdaten angebracht ist, bzw. neue Indexe aufgebaut werden sollten etc. Das Erzeugen der Daten eines Ausführungsplans kann unter Oracle mit der Anweisung

```
EXPLAIN PLAN FOR <Anweisung>;
```

erfolgen. `<Anweisung>` wird an den Anfrageoptimierer übergeben. Dieser ermittelt den Ausführungsplan und schreibt die Planinformationen in die Tabelle `PLAN_TABLE`. Diese Tabelle muss zuvor im Schema des Benutzers angelegt werden, der `EXPLAIN PLAN` ausführen will. Dazu ist das im Lieferumfang von Oracle enthaltene Skript `utlxplan.sql` auszuführen. *Abbildung 64* zeigt die Erzeugung von Planinformationen unter Oracle.

```
SQLWKS> delete from plan_table;
0 Zeilen verarbeitet

SQLWKS> explain plan for
     2>         SELECT /*+ INDEX_ASC(ABTEILUNG I_ANR) NOCACHE */ *
        FROM abteilung where anr>=0 and anr<=100000 ORDER BY anr;
Anweisung verarbeitet

SQLWKS> SELECT operation, options,object_name,id,cost,parent_id,position
     2>  FROM plan_table;

OPERATION         OPTIONS       OBJECT_NAME  ID    COST  PARENT_ID  POSITION
----------------- ------------- ------------ ----- ----- ---------- --------
SELECT STATEMENT                             0     52                1
TABLE ACCESS      BY INDEX ROWID ABTEILUNG    1     50      0         1
INDEX             RANGE SCAN     I_ANR        2      2      1         1
3 Zeilen ausgewählt
```

Abbildung 64: Beispiel von Planinformationen unter Oracle

Administrationswerkzeuge bieten typischerweise auch grafisch aufbereitete Darstellungen von Ausführungsplänen an (*Abbildung 65* und *Abbildung 66*).

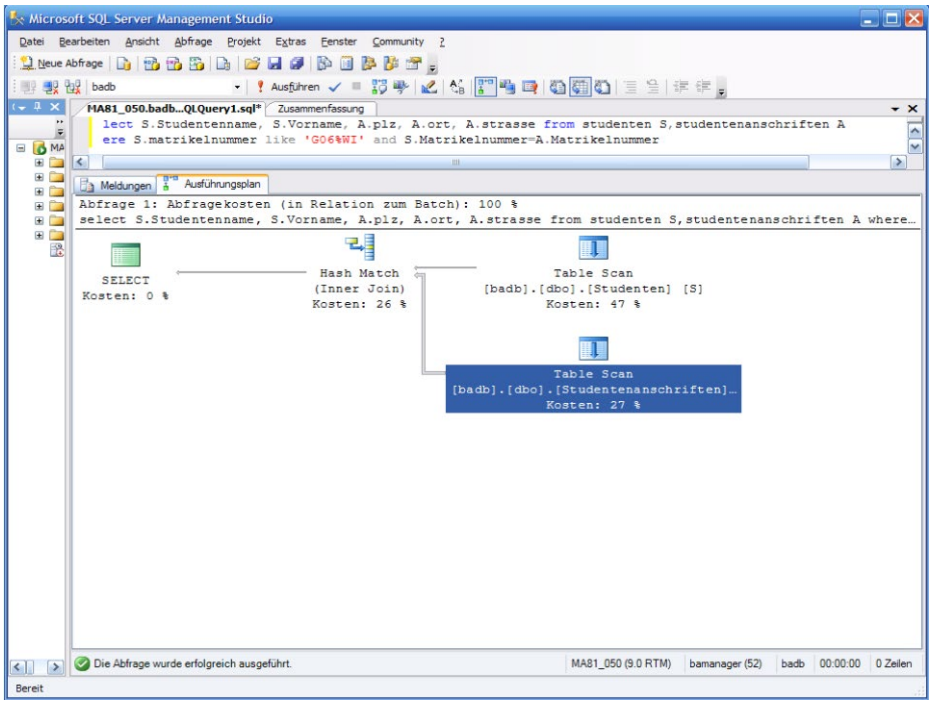


Abbildung 65: Beispiel für die grafische Darstellung eines Ausführungsplans

Die beiden Abbildungen wurden mit dem Enterprise Manager für den Microsoft SQL Server erzeugt. Aber auch die Pendants von Oracle, DB2 usw. bieten entsprechende Darstellungen.

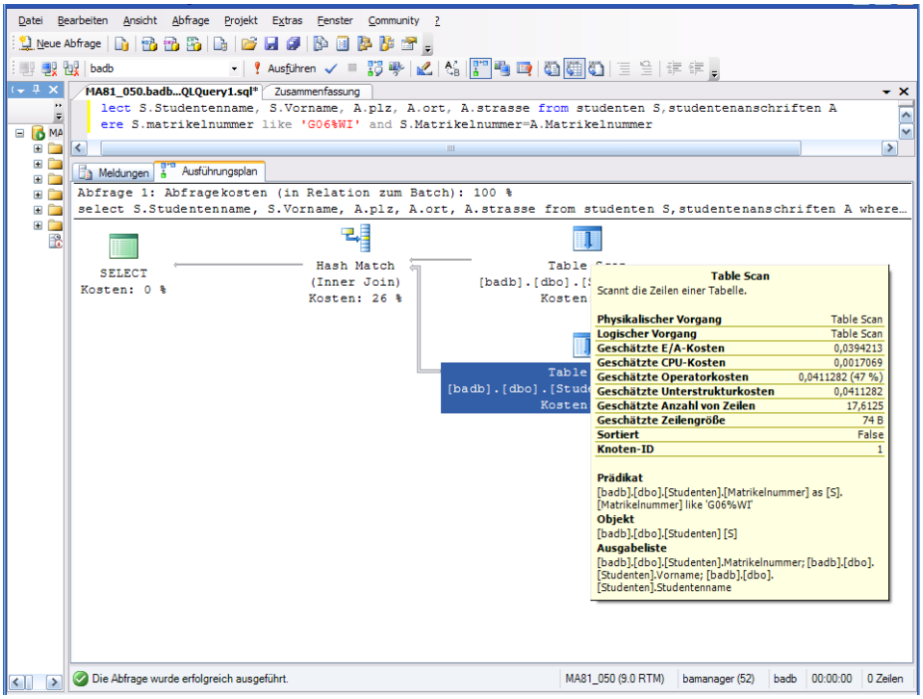


Abbildung 66: Ausführungsplan mit Informationen zu einem Planoperator

Oracle bietet mit sog. *Optimizer Hints* die Möglichkeit, das Verhalten des Anfrageoptimierers zu beeinflussen. Dieser versucht, die Hinweise bei der Anfrageoptimierung zu befolgen. Damit ist eine Steuerung des Optimierungsvorgangs möglich, mit der festgelegt werden kann, wie auf Daten

zuzugreifen ist. Voraussetzung für die Befolgung ist, dass die Zugriffe prinzipiell möglich sind. Soll ein Zugriff über einen Index erfolgen, muss der entsprechende Index natürlich vorhanden sein. Die Syntax lautet wie folgt:

```
{DELETE|INSERT|SELECT|UPDATE}
    /*+ <hint> [<text>][<hint> [<text>]]... */ ...
```

Beispiel:

```
SELECT /*+ INDEX_ASC(ABTEILUNG I_ANR) NOCACHE */ *
FROM abteilung where anr>=0 and anr<=100000 ORDER BY anr;
```

Tabelle 5 zeigt einige Beispiele für Optimierer-Hinweise.

Hint	Funktion
`/*+ RULE */`	aktiviert regelbasierte Optimierung
`/*+ CLUSTER(table) */`	Cluster-Scan auf Tabelle
`/*+ FULL(table) */`	Full Table Scan auf Tabelle
`/*+ HASH(table) */`	Hash Scan auf Tabelle
`/*+ INDEX_ASC(table index) */`	Index-Bereichs-Scan auf Tabelle
`/*+ USE_NL (table) */`	Nested Loop Join mit table als innere Tabelle
`/*+ NOCACHE */`	Einordnung der gelesenen Blöcke am Ende der LRU-Liste
`/*+ CACHE */`	Einordnung der gelesenen Blöcke am Anfang der LRU-Liste

Tabelle 5: Beispiele für Hints

7.3 Kostenschätzungen

In diesem Abschnitt soll an einigen einfachen Beispielen die Schätzung der von den einzelnen Planoperatoren verursachten Ausführungskosten betrachtet werden. Zur Vereinfachung werden bei Index Lookup und Index Range Scan nur eindeutige (unique) und Non Clustered Indexe betrachtet. Weiterhin werden Funktionen zur Kosten- und Mehraufwandsabschätzung für sequenzielles Suchen und Zugriffe auf die zu einem Cluster-Schlüsselwert gehörenden Tupel vorgestellt. Die Basis für die Kostenschätzungen bilden zunächst bekannte Kostenfunktionen [SAC+79]. Die verwendeten Formelzeichen werden an entsprechender Stelle erläutert.

Für den Datenzugriff über einen Index (Index Lookup) kann die anfallende Anzahl Blockzugriffe (C_L) mit der Gleichung

$$C_L = I_{LEV} + 1 \qquad (1)$$

abgeschätzt werden, wenn aus den Statistikdaten die Anzahl Ebenen des Index (I_{LEV} - im Beispiel in Abbildung 46 und Abbildung 48 ist diese jeweils gleich 2) ermittelt werden kann. Soll berücksichtigt werden, dass auch migrierte Tupel (Satzauslagerungen) vorkommen können, so muss die Gleichung um die Gesamtzahl der Tupel der Tabelle (T) und die Anzahl ausgelagerter Tupel/Sätze (A) erweitert werden.

$$C_L = I_{LEV} + 1 + \frac{A}{T} \tag{2}$$

Viele der üblichen Kostenmodelle berücksichtigen Pufferungsgrade (Buffer Hit Ratio) von Daten- und Indexblöcken entweder gar nicht oder mittels grober Annahmen. Dies ist oft darauf zurückzuführen, dass insbesondere detaillierte datenbankobjekt- und operationsbezogene Statistiken (etwa tabellen- oder indexbezogen) über Pufferungsgrade von DBMS-Produkten derzeit i.d.R. nicht direkt bzw. nicht nach „außen“ angeboten werden, weil sie sehr feingranular sind und übergreifend über mehrere Schichten der DBMS-Architektur gesammelt werden müssten. Eine entsprechende Erweiterung der Kostenfunktionen um datenbankobjektbezogene Pufferungsgrade, also die *Wahrscheinlichkeit, dass sich ein angeforderter Block im Puffer-Pool des DBMS befindet*, würde zu folgender Gleichung für die Berechnung der Anzahl anfallender physischer Blockzugriffe (C_P) führen.

$$C_P = \underbrace{\sum_{i=1}^{I_{LEV}} \left(1 - I_{BRi}\right)}_{Indexteil} + \underbrace{\left(1 + \frac{A}{T}\right) * \left(1 - T_{BR}\right)}_{Datenteil} \tag{3}$$

Dabei stehen I_{BRi} und T_{BR} für die Pufferungsgrade von Index- und Datenblöcken. Hier fließt implizit die gängige Annahme ein, dass Zugriffe auf Datenblöcke im Puffer, im Vergleich zu Zugriffen auf Sekundärspeichermedien, kostenmäßig vernachlässigt werden können. Bei der Berechnung des Aufwands zum Durchsuchen des Indexteils wird für jede Ebene (i) des Indexbaums, beginnend bei der Wurzel, ein eigener Pufferungsgrad berücksichtigt, da bei Operationen über Indexen physische I/O-Vorgänge meist nur für Blöcke auf der Blattebene anfallen. Die Blöcke der inneren Ebenen werden mit hoher Wahrscheinlichkeit im Datenbankpuffer vorgehalten. Ist eine Ermittlung von ebenenbezogenen Pufferungsgraden für Indexe in einer konkreten Systemumgebung nicht möglich, so können hier zumindest auch entsprechende Annahmen in die Berechnungen einfließen.

Als zweiter Beispieloperator soll ein Index Range Scan betrachtet werden. Die Anzahl notwendiger Blockzugriffe ergibt sich hier aus der Summe der Anzahl Indexebenen (ohne Blattebene), der Anzahl Blätter des Indexbaums, die entlang der Verkettung durchlaufen werden müssen (I_{LEAF} bzw. ein Teil davon) und der Anzahl Tupel im entsprechenden Bereich (T), da für jeden Tupelzugriff ein Zugriff auf den jeweiligen Datenblock erfolgen muss.

$$C_L = \left\lceil \underbrace{\left(I_{LEV} - 1\right)}_{innereEbenen} + \underbrace{\max\left(S * I_{LEAF}, 1\right)}_{Blattebene} + \underbrace{T * S}_{Datenteil} \right\rceil \tag{4}$$

Hier gibt S (Selektivität) den Anteil des Suchbereichs an der Gesamtdatenmenge an, über den der Index Scan ausgeführt wird. Dieser kann bei einer Intervallanfrage im einfachsten Fall mit der folgenden Gleichung näherungsweise berechnet werden.

$$S = \frac{1 + SK_{MAX} - SK_{MIN}}{1 + I_{MAX} - I_{MIN}}, \tag{5}$$

- SK_{MAX} entspricht dabei der oberen Intervallgrenze in der Selektionsbedingung.
- SK_{MIN} entspricht der unteren Intervallgrenze in der Selektionsbedingung.
- I_{MAX} entspricht dem größten Schlüsselwert des Index.
- I_{MIN} entspricht dem kleinsten Schlüsselwert des Index.

Voraussetzung für dieses sehr einfache Verfahren ist in etwa eine Gleichverteilung der Schlüsselwerte. DBMS-Produkte benutzen ansonsten intern oft Histogramme, um genauere Informationen über Werteverteilungen oder gar Wertekorrelationen festzuhalten.

Durch die Berücksichtigung der durch ausgelagerte Sätze zusätzlich anfallenden Blockzugriffe (z.B. Zugriff Nr. 4 in Abbildung 49) ergibt sich folgende Gleichung.

$$C_L = \lceil (I_{LEV} - 1) + \max(S * I_{LEAF}, 1) + (T + A) * S \rceil \tag{6}$$

Eine Erweiterung um Pufferungsgrade führt dann zu Gleichung 7. Dabei entspricht I_{BRH} dem Pufferungsgrad der Blöcke auf der Blattebene.

$$C_P = \left\lceil \underbrace{\sum_{i=1}^{I_{LEV}-1} (1 - I_{BRi})}_{innereEbenen} + \underbrace{\max(S * I_{LEAF} * (1 - I_{BRH}), (1 - I_{BRH}))}_{Blattebene} + \underbrace{(T + A) * (1 - T_{BR}) * S}_{Datenteil} \right\rceil \tag{7}$$

Als Wert für die bei einer sequenziellen Suche (Table Scan) anfallende Anzahl Blockzugriffe wird üblicherweise die Anzahl von der Tabelle aktuell belegter Blöcke ($P_{AKTUELL}$) verwendet, unabhängig von deren Füllungsgrad. Hier ist auch in gängigen Kostenmodellen der durch evtl. vorhandenen eingestreuten Freiplatz entstehende Mehraufwand somit implizit bereits enthalten. Die Kosten für einen Table Scan können wie folgt bestimmt werden.

$$C_P = P_{AKTUELL} * (1 - T_{BR}) \tag{8}$$

Sollen alle zu einem Cluster-Schlüsselwert gehörenden Tupel gelesen werden, so fallen bei einem Index Cluster die Zugriffe für das Durchmustern des Indexbaums, ein Zugriff auf den Datenblock im Primärbereich und evtl. die Zugriffe für das Durchsuchen der Überlaufbereiche an. Der I/O-Aufwand kann mit der Gleichung

$$C_L = \underbrace{I_{LEV}}_{Indexteil} + 1 + \underbrace{\frac{I_{CHAINS}}{I_{UNQ}}}_{Überlaufbereich} \tag{9}$$

abgeschätzt werden. Die Anzahl Cluster-Schlüsselwerte fließt über I_{UNQ} und die Gesamtzahl der zum Cluster gehörenden Überlaufblöcke über I_{CHAINS} ein. Die Berücksichtigung von Pufferungsgraden führt zur folgenden Gleichung.

$$C_P = \underbrace{\sum_{i=1}^{I_{LEV}} (1 - I_{BRi})}_{Indexteil} + \underbrace{\left(1 + \frac{I_{CHAINS}}{I_{UNQ}}\right) * (1 - T_{BR})}_{Datenteil} \tag{10}$$

8 Benutzerverwaltung

In diesem Kapitel werden kurz einige Aspekte betrachtet, die in Zusammenhang mit der Verwaltung von Zugriffsrechten für Datenbanken stehen. Für weitere Informationen, besonders zu Syntax und Semantik der in den Beispielen gezeigten Anweisungen sei hier ausdrücklich auf die jeweilige Systemliteratur verwiesen.

Das Ziel der Benutzerverwaltung ist eine Reglementierung des Zugriffs auf Datenbankobjekte und -inhalte. Zum Aufbau einer Verbindung mit einem Datenbanksystem ist mindestens ein *Benutzerkonto* nötig. Beim Anlegen eines Benutzerkontos können diesem unter Oracle z.B.

- ein Standard-Table-Space,
- ein Table Space für temporäre Tabellen etc.,
- Begrenzungen für Speicherplatzreservierungen in Table Spaces und
- Profile, die Limitierungen von Ressourcen enthalten,

zugewiesen werden.

Beispiel:

```
CREATE USER meier IDENTIFIED BY huhu
DEFAULT TABLESPACE nutzer_ts
QUOTA 10M ON nutzer_ts
QUOTA 5M ON temp_ts
QUOTA 5M ON system
PROFILE def_user
PASSWORD EXPIRE;
```

Den Benutzerkonten können unter Oracle noch Benutzerprofile zugeordnet werden. Ein Benutzerprofil ist dabei eine Menge von Limitierungen für die Ressourcennutzung. Wird einem Benutzer ein Profil zugeordnet, so kann er diese Limitierungen nicht überschreiten.

Beispiel:

```
CREATE PROFILE administrator LIMIT
SESSIONS_PER_USER UNLIMITED
CPU_PER_SESSION UNLIMITED
CPU_PER_CALL 3000
CONNECT_TIME 45
LOGICAL_READS_PER_CALL 1000
PRIVATE SGA 15K
PASSWORD_LIFE_TIME 60
PASSWORD_REUSE_TIME 60
PASSWORD_REUSE_MAX UNLIMITED;
```

Zugriffsrechte können benutzer- oder aufgabenbezogen vergeben werden. Bei der *benutzerbezogenen Rechtevergabe* werden Zugriffsrechte und –privilegien direkt an einzelne Benutzerkennungen vergeben, mit denen sich die Nutzer dann beim DBMS ausweisen können. Diese individuelle Vergabe ist in kleinen Organisationen leicht zu realisieren. Bei der *aufgabenbezogenen Rechtevergabe* werden Zugriffsrechte und -privilegien nicht an einzelne Benutzer(kennungen) vergeben, sondern an Datenbankrollen. Eine Datenbankrolle ist eine benannte Menge von Zugriffsrechten. Diese werden nach Aufgabengebieten vergeben. Die Rollen werden dann einzelnen Datenbanknutzern zugewiesen, welche dadurch die Rechte der Rolle erhalten. Durch die Nutzung von Rollen ergeben sich in größeren Organisationen Vereinfachungen weil:

- mehreren Benutzern mit gleichem Aufgabengebiet nur die gleiche Rolle zugewiesen werden muss. Bei Rechteänderungen für diese Gruppe ist nur die Änderung bei der Rolle, nicht bei den einzelnen Mitgliedern, nötig.
- bei Aufgabenwechsel von Datenbankbenutzern nur ein Rollenwechsel vorzunehmen ist.
- ein Sicherheitskonzept aufgabenbezogen und unabhängig von konkreten Personen aufgebaut werden kann.

Nach dem Anlegen ist eine Rolle zunächst leer. Die Rechte werden wie üblich mit `GRANT` vergeben und mit `REVOKE` entzogen.

Beispiel:

```
CREATE ROLE normaler_nutzer;
```

Die Zuordnung eines Nutzers zu einer Rolle erfolgt z.B. mit der Anweisung `ALTER USER` oder mit der Anweisung `GRANT`.

Beispiele:

```
ALTER USER meier DEFAULT ROLE normaler_nutzer;
GRANT normaler_nutzer TO schulze;
```

Rechte können an Datenbankbenutzer (Benutzerkonten) und Rollen vergeben werden. Rollen können bei der Vergabe von Rechten wie Systemprivilegien verwendet werden. Oracle unterscheidet (wie viele DBMS-Produkte) in Objektrechte und Systemprivilegien. *Objektrechte* regeln den Zugriff auf Datenbankobjekte (Tabellen, Sichten etc.). Sie dienen z.B. auch zur Umsetzung von Datenschutzkonzepten.

Beispiele:

```
GRANT UPDATE (name,strasse,ort,plz) ON personal TO meier;
GRANT SELECT ON personal TO normaler_nutzer;
```

Systemprivilegien regeln die Berechtigung, SQL-Anweisungen zu nutzen.

Beispiel:

```
GRANT CREATE TABLE TO mueller;
```

Das Entziehen von Rechten erfolgt mit der Anweisung `REVOKE`. `REVOKE` ist wie `GRANT` aufgebaut, aber statt `GRANT ... TO ...` wird `REVOKE ... FROM ...` angegeben. `REVOKE` dient zum Entziehen von Objektrechten und Systemprivilegien.

Beispiele:

```
REVOKE DROP ANY TABLE FROM meier,schulze;

REVOKE DELETE ON personal FROM meier;

REVOKE ALL ON personal FROM schmidt;

REVOKE UPDATE ON berichte FROM public;
```

Die entsprechenden Aufgaben der Benutzerverwaltung und Rechtevergabe können oft über Administrationswerkzeuge realisiert werden, die meist über grafische Benutzungsschnittstellen verfügen (*Abbildung 67*). Die Benutzung von solchen Werkzeugen ist i.d.R. bequemer als die Benutzung von Kommandos. Allerdings sollte dringend darauf geachtet werden, dass alle mit solchen Werkzeugen vorgenommenen Maßnahmen gut dokumentiert werden sollten, damit sie später bei Bedarf nachvollzogen werden können.

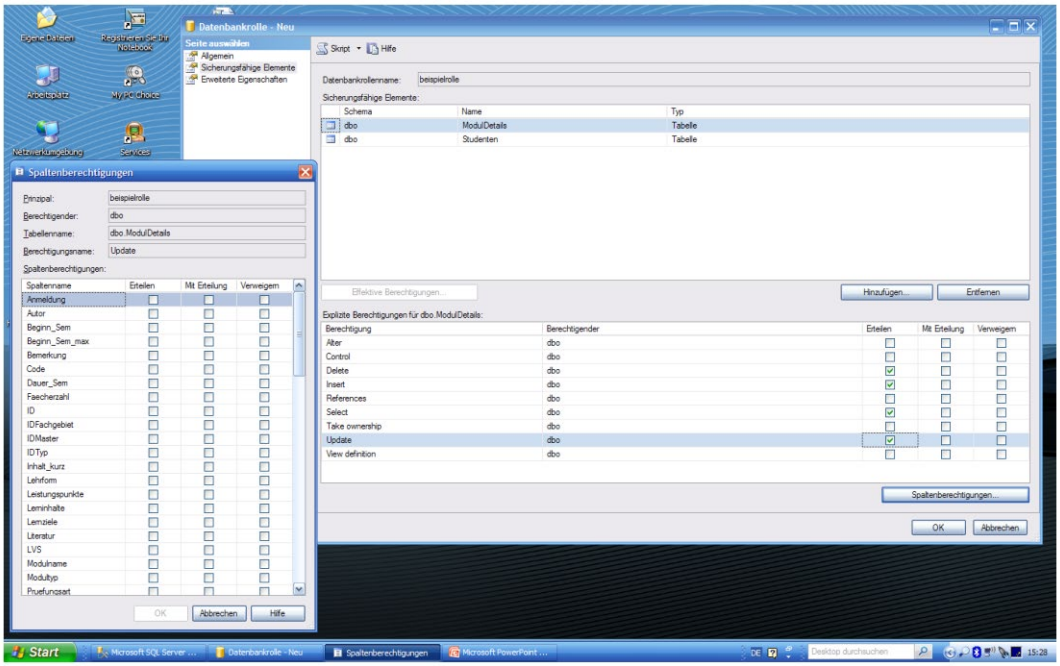


Abbildung 67: Vergabe von Zugriffsrechten mittels Administrationswerkzeugen

Literaturverzeichnis

[Che76] P.P. Chen. The entity-relationship model – toward a unified view of data. In *ACM TODS, Vol. 1, No. 1*, New York, 1976

[Cod71] E. F. Codd. Further Normalization of the Data Base Relational Model. IBM Research Report, San Jose, California, 1971

[GLP+76] J.N. Gray, R.A. Lorie, G.B. Putzolu, I.L. Traiqer. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *Proceedings of the IFIP Working Conference on Modelling of Database Management Systems*. Freudenstadt, Deutschland, Januar 1976

[GR93] J. Gray, A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publ., 1993

[HR01] T. Härder, E. Rahm. *Datenbanksysteme - Konzepte und Techniken der Implementierung*, 2. überarb. Auflage. Springer-Verlag, Berlin/ Heidelberg/New York, 2001

[Keß95] U. Keßler. *Flexible Speicherungsstrukturen und Sekundärindexe in Datenbanksystemen für komplexe Objekte.* Dissertation. Fakultät für Informatik, Universität Ulm, Mai 1995

[IBM02] *IBM DB2 Universal Database Command Reference Version 8*. International Business Machines Corporation, 2002

[IBM04] *DB2 UDB V8 and WebSphere V5 Performance Tuning and Operations Guide,* First Edition. International Business Machines Corp., März 2004

[Kud07] T. Kudraß (Hrsg.). Tachenbuch Datenbanken. Fachbuchverlag Leipzig im Carl Hanser Verlag, Leipzig, 2007

[Ora03a] *Oracle Database Administrator's Guide 10g Release 1 (10.1)*. Oracle Corporation, Dezember 2003

[Ora03b] *Oracle Database Concepts 10g Release 1 (10.1).* Oracle Corporation, Dezember 2003

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, T. Price. Access Path Selection in a Relational Database Management System. In *Proc. Of the ACM SIGMID Conference*. 1979.

[SHS05] G. Saake, A. Heuer, K.-U. Sattler. *Datenbanken: Implementierungstechniken* (2., aktualisierte und erweiterte Auflage). mitp-Verlag, Bonn, 2005

[Ska06] S. Skatulla. *Speicherung und Indexierung komplexer Objekte in objektrelationalen Datenbank-Management-Systemen.* Dissertation. Institut für Informatik, Friedrich-Schiller-Universität Jena, April 2006

[SST97] G. Saake, I. Schmitt, C. Türker. *Objektdatenbanken - Konzepte, Sprachen, Architekturen*. International Thomson Publishing, Bonn, 1997

[TL91] R. Trautloft, U. Lindner. *Datenbanken - Entwurf und Anwendung.* Verlag Technik, Berlin, 1991

[ZSL98] C. Zou and B. Salzberg and R. Ladin. Back to the Future: Dynamic Hierarchical Clustering. In *Proc. of the 14th International Conference on Data Engineering.* Orlando, USA, Februar 1998

Anhang A - Beispielprogramm ESQL/C

```
# include <stdio.h>
# include <string.h>

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
     char v_g_plz[7],v_g_name[51];
     VARCHAR u_name[81];
EXEC SQL END DECLARE SECTION;

void meldung(void);
void meldung()
{
     char buf[200];
     int bufsize=199;
     int length;

     sqlglm(buf,&bufsize,&length);
     printf("\n%s\n",buf);
}

int main(void)
{
     EXEC SQL WHENEVER SQLWARNING DO meldung();
     EXEC SQL WHENEVER SQLERROR DO meldung();
     printf("\nLogin in folgender Form angeben:\n");
     printf("<Benutzerkennung>/<Kennwort>@<Server>\n");
     printf("Login: ");
     gets(u_name.arr);
     u_name.len=strlen(u_name.arr);
     EXEC SQL CONNECT  :u_name;
     if(sqlca.sqlcode!=0)
     {
          printf("Connect nicht möglich \n");
          return(-1);
     }
     printf("Connect erfolgreich \n");
     printf("\nAb welcher Postleitzahl soll angezeigt werden ? ");
     gets(v_g_plz);
     EXEC SQL  DECLARE c_sel CURSOR FOR
          SELECT g_plz,g_name FROM gemeinde
          WHERE g_plz >= :v_g_plz;
     EXEC SQL OPEN c_sel;
     if(sqlca.sqlcode==0)
          printf("\nPostleitzahl   Gemeindename\n");
     else{
          printf("\nKeine Daten gefunden oder Fehler bei der
          Verarbeitung\n");
          EXEC SQL COMMIT WORK RELEASE;
          return(-2);
     }
     while(1){
          EXEC SQL FETCH c_sel INTO :v_g_plz,:v_g_name;
          if(sqlca.sqlcode!=0)
               break;
          printf("\n%s   %s",v_g_plz,v_g_name);
     }
     EXEC SQL CLOSE c_sel;
     EXEC SQL COMMIT WORK RELEASE;
     printf("\nProgramm beendet !\n");
}
```

Anhang B - Beispielprogramm Java und JDBC

```
import java.sql.*;
import java.util.Scanner;

public class modelliste  // Anwendungsklasse
{

     String hersteller="Opel";     // Stringvariable fuer Herstellerauswahl
     static Connection conDB;      // Verbindungskontext
     static PreparedStatement st_select, st_insert; // Anweisungen fuer Abfrage- und
                                                       Einfuegeoperation

     public void work() // ruft die Datenbankfunktionen auf
     {
          try
          {
               connectDB();
               runSelect();
               insertCar();
               runSelect();

          }
          catch (Exception e) // Ausnahmebehandlung
          {
               System.out.println("Fehler:"+e.getMessage());
          }
     }

     public void connectDB()  // DB-Verbindungsaufbau und Vorbereitung der
                              Anweisungen
     {

          try
          {
               //   Treiberauswahl und Verbindungsaufbau
               Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

               conDB = DriverManager.getConnection("jdbc:sqlserver://192.168.71.1;
                         databaseName=WS1;User=WS1;password=MSsqlWS_1");

               System.out.println("Verbunden");

               // Vorbereiten der Anweisungen
               st_select=conDB.prepareStatement("SELECT fznr,typ,kennzeichen
                              FROM bestand WHERE hersteller=?");
               st_insert=conDB.prepareStatement("INSERT INTO bestand
                              VALUES (?,?,?,?,?,?)");

               // Commits setzt die Anwendung
               conDB.setAutoCommit(false);
          }
          catch (Exception e) // Ausnahmebehandlung
          {
               System.out.println("Fehler:"+e.getMessage());
          }
     }
```

```
public void runSelect()  // Erzeugt eine Liste allse Fahrzeuge eines Herstellers
{
     ResultSet selset;        // Container fur Treffermenge
     String typ,kennzeichen;  // Variablen zur Datenaufnahme
     int fznr;                // Variable zur Datenaufnahme

     // Einlesen des Namens des Herstellers
     Scanner in = new Scanner(System.in);
     hersteller = in.nextLine();
     in.close();

try
     {
          st_select.setString(1,hersteller);   // Setzen des Auswahlkriteriums
          selset=st_select.executeQuery();     // Ausfuehren der Abfrage

          System.out.println(" Fahrzeug-Nr.\t| Typ\t\t\t| Kennzeichen");
          System.out.println("-----------------------------------------------");

          while(selset.next())// Holen der einzelnen Tupel der Treffermenge
          {
               // Uebertragen der Elemente eines Tupels
               fznr=selset.getInt(1);
               typ=selset.getString(2);
               kennzeichen=selset.getString(3);

               // "Verarbeitung" (Ausgabe) der Daten des Tupels
               System.out.println(" "+fznr+"\t| "+typ+"\t| "+kennzeichen);
          }
          selset.close();
     }
     catch (Exception e)      // Ausnahmebehandlung
     {
          System.out.println("Fehler:"+e.getMessage());
     }

}

public void insertCar()       // Fuegt ein neues Fahrzeug ein
{
     ResultSet rs;  // Container fuer groeßsste Fahrgestellnr.
     int nextfznr;  // Variable zur Aufnahme der Fahrgestellnummer

     try
     {
          // Ermittlung der bisherigen groessten Fahrgestellnummer
          // Erzeugen einer Anweisung
          Statement anwsg=conDB.createStatement();

          // Abfrage der bish. groessten Fahrgestellnr.
          rs=anwsg.executeQuery("SELECT MAX(fznr) FROM bestand");
          rs.next();               // Holen des Tupels
          nextfznr=rs.getInt(1);   // Uebertragen der Fahrgestellnummer
          rs.close();
          nextfznr++;

          // Setzen der Werte fuer die INSERT-Anweisung
          st_insert.setInt(1,nextfznr);        // Fahrgestellnr.
          st_insert.setString(2,"Opel");       // Hersteller
          st_insert.setString(3,"Fantasia");   // Typ
          st_insert.setDate(4,new java.sql.Date(2222222));
          st_insert.setString(5,"G-AB 123");   // Kennzeichen
          st_insert.setInt(6,2);               // Filiale

          st_insert.executeUpdate();    // Ausfuehren der INSERT-Anweisung
          conDB.commit();               // Transaktion abschliessen
     }
     catch (Exception e) // Ausnahmebehandlung
     {
          System.out.println("Fehler:"+e.getMessage());
     }
}
```

```
public static void main(String [] args) // Hauptfunktion
     {
          try
          {
               // Erzeugen einer Instanz der Anwendungsklasse
               modelliste appl=new modelliste();

               // Start der Anwendungsfunktion
               appl.work();
          }
          catch (Exception e)
          {
               System.out.println("Fehler:"+e.getMessage());
          }
     }
}
```