

Datenbanken I

(T2INF2004)

Lehrbeauftragter

Wolfgang Stark

wstark@lehre.dhbw-stuttgart.de

Wolfgang Stark

Inhaltsverzeichnis

1	Grundlagen und Begriffe	8
1.1	Was sind Datenbanken?.....	8
1.2	Geschichte.....	9
1.2.1	Wie Wissen entsteht	9
1.2.2	Wie entstand die erste Datenverarbeitung?	9
1.2.3	Elektronische Informationsablage	10
1.3	Zeitliche Entwicklung der verschiedenen Datenmodelle	11
1.3.1	Das hierarchische Datenmodell	11
1.3.2	Das Netzwerk Datenmodell.....	12
1.3.3	Das Relationale Datenmodell	13
1.4	Die Codd'schen Regeln	13
1.5	Die wichtigsten Begriffe kurz erklärt	15
1.6	Die 3-Ebenen Architektur.....	16
1.7	Die Systemarchitektur	17
1.8	Von der Idee zur Datenbank (Entwurfsprozess).....	18
2	Der konzeptionelle Datenbankentwurf.....	20
2.1	Anforderungsanalyse	20
2.2	Zielsetzung.....	22
2.3	Das Softwarehaus	22
2.4	Die Komponenten des ERM	23
2.5	Die Relationship	23
2.6	Die Attribute und Werte	24
2.7	Weitere Attribute	27
2.8	Kardinalitäten	28
2.9	Weitere Notationen.....	29
2.9.1	Krähenfußnotation	29
2.9.2	Die Min-Max-Notation.....	30
2.10	Besondere Beziehungen.....	31
2.11	Schwache Entitäts-Typen	34
2.12	Sichtenkonsolidierung	34
3	Der Relationale Entwurf	36
3.1	Was versteht man unter einer Relation	36
3.1.1	Theoretische Betrachtung	36
3.1.2	Von der Relation zur Tabelle.....	37
3.1.3	Domänen, Schema und Instanzen.....	38
3.2	Umsetzung ER- auf Relationales-Modell	39
3.2.1	Überführung von Entitäts-Typen in Relationen	40
3.2.2	Umsetzung schwacher Entitäts-Typen	40
3.2.3	Umsetzung von Beziehungs-Typen in Relationen	40
3.3	Daten- und Referentielle Integrität	46
4	Die relationale Entwurfstheorie (Normalisierung)	47
4.1	Anomalien.....	47
4.2	Funktionale Abhängigkeit	49
4.3	Interferenzregel und Armstrong-Axiome	49
4.4	Schlüssel	50
4.5	Normalformen.....	51
4.5.1	1. Normalform	51

4.5.2	2. Normalform	52
4.5.3	3. Normalform	53
4.5.4	Boyce-Codd Normalform	54
4.5.5	Von 3NF nach Boyce-Codd und zurück.....	55
4.5.6	4. und 5. Normalform	56
4.5.7	Verlustfreie und abhängigkeitsbewahrende Zerlegung	56
5	SQL-Teil1	58
5.1	Einführung	58
5.1.1	Historie	58
5.1.2	SQL-Standards.....	58
5.1.3	PostgreSQL - Das System zum Üben.....	59
5.1.4	Bestandteile von SQL	60
5.2	Datenbanken und Tabellen definieren (DDL)	61
5.2.1	CREATE Database	61
5.2.2	CREATE Schema	62
5.2.3	Tabellen anlegen und verwalten	62
5.2.3.1	CREATE Table.....	62
5.2.3.2	ALTER Table	66
5.2.3.3	DROP.....	66
5.3	Daten in Tabellen verändern (DML)	67
5.3.1	Eine neue Zeile einfügen	67
5.3.2	Bestehende Daten verändern	68
5.3.3	Löschen von Daten	68
5.3.4	Daten in verbundenen Tabellen löschen.....	68
5.3.4.1	Abhängige Daten auf NULL setzen	69
5.3.4.2	Die abhängigen Daten werden mit gelöscht	69
5.3.4.3	Löschen der abhängigen Daten abweisen.....	69
5.3.4.4	Hinweise zu Löschregeln.....	70
5.3.4.5	Regeln für den Update	70
6	Die Relationale Algebra (eine formale Sprache).....	71
6.1	Definition	71
6.2	Selektion	71
6.3	Projektion.....	72
6.4	Vereinigung (U).....	73
6.5	Differenz (-)	74
6.6	Kreuzprodukt (x)	75
6.7	Umbenennung (ρ)	76
6.8	Durchschnitt (\cap)	77
6.9	Theta-Join (\bowtie_p)	79
6.10	Equi-Join ($\bowtie_{[L],[R]}$)	80
6.11	Natürliche Join (\bowtie)	80
7	SQL Teil 2	82
7.1	Daten abfragen (DRL)	82
7.1.1	Einfache Abfrage von Daten	82
7.1.1.1	Distinct und Order By.....	83
7.1.1.2	Wertelisten und flexibles Suchen	84
7.1.1.3	Funktionen	84
7.1.1.4	Gruppenbildung	85
7.1.1.5	Zusammenfassung des SELECT	86
7.1.2	Unterabfragen	86

7.1.3	Mehrere Abfragen miteinander verknüpfen	89
7.1.3.1	UNION (Vereinigung).....	89
7.1.3.2	INTERSECT (Durchschnitt)	89
7.1.3.3	EXCEPT (Differenz)	89
7.1.4	Abfragen auf mehrere Tabellen	90
7.1.4.1	Verknüpfung zweier Tabellen	90
7.1.4.2	Der INNERE JOIN	90
7.1.4.3	Der ÄUSSERE JOIN	92
7.2	Views in SQL	94
7.2.1	Einfache Sichten	94
7.2.2	Datenunabhängigkeit durch Views.....	95
7.2.3	Über Sichten Daten verändern.....	95
8	Das Transaktionskonzept (TCL).....	97
8.1	Was versteht man unter einer Transaktion	97
8.2	Verwaltung einer Transaktion	97
8.2.1	Befehle zur Steuerung einer Transaktion	97
8.2.2	ACID.....	98
8.3	Das Protokoll einer Transaktion	98
8.4	Der Online-Shop	99
8.4.1	Erweiterung des ERM.....	99
8.4.2	Warenbestellungen	99
8.4.3	Ablauf einer Transaktion	101
8.5	Konkurrierende Zugriffe.....	102
8.5.1	Voraussetzung.....	102
8.5.2	Beispiele konkurrierender Zugriffe	103
8.5.3	Isolationsstufen	104
9	Indizierung von Tabellen.....	107
9.1	Speicher- und Zugriffsmöglichkeit von Daten	107
9.1.1	Der sequentielle Zugriff.....	107
9.1.2	Der direkte Zugriff.....	107
9.1.3	Index-sequentieller Zugriff.....	107
9.1.4	Baumstruktur	108
9.1.5	Schlüsselbaum (B-Baum)	108
9.2	Indizes.....	110
9.2.1	Index auf Primärschlüssel.....	110
9.2.2	Verwendung weiterer Indizes	110
9.2.3	Erstellen eines Index.....	110
10	Schlusswort.....	114
11	Literaturverzeichnis	115
12	Links und Referenzen	117
12.1	Links	117
12.2	Datentypen	118
13	Index	119

Abbildungsverzeichnis

Abbildung 1: Datenbanksystem.....	9
Abbildung 2: Abbildung eines Magnetbandlaufwerks	11
Abbildung 3: Hierarchische Datenmodell	11
Abbildung 4: Abbildung einer komplexen Beziehung in einem Netzwerk-Datenmodell.....	13
Abbildung 5: Beispiel für eine Benutzersicht.....	14
Abbildung 6: Architekturübersicht DBMS	17
Abbildung 7: Datenbank-Entwurfsprozess	19
Abbildung 8: Verschiedene Dokumentationstechniken	21
Abbildung 9: Entitäten Softwarehaus.....	23
Abbildung 10: Beziehung Mitarbeiter-Projekt	24
Abbildung 11: Entität mit Attributen.....	25
Abbildung 12: Entität mit Schlüsselattributen.....	26
Abbildung 13: Alle Entitäts-Mengen des Softwarehauses mit Schlüsselattributen	27
Abbildung 14: Modellierung von Mengenattributen.....	27
Abbildung 15: 1:1-Beziehung	28
Abbildung 16: 1:N-Beziehung.....	28
Abbildung 17: M:N-Beziehung	29
Abbildung 18: Attribute bei Beziehungsmengen	29
Abbildung 19: Gegenüberstellung Chen-/Krähenfuß-Notation	29
Abbildung 20: Modellierung mit Krähenfußnotion.....	30
Abbildung 21: (min,max)-Notation vs. Chen	30
Abbildung 22: Drei-stellige Beziehung	31
Abbildung 23: Drei-stellige Beziehung mit Kardinalitäten.....	31
Abbildung 24: Rekursive Beziehung.....	32
Abbildung 25: Ist-ein- Beziehung	32
Abbildung 26: Ist-ein- Beziehung mit Beziehung.....	33
Abbildung 27: Partitionierung am Beispiel Auto	34
Abbildung 28: Schwache Entitäts-Typen.....	34
Abbildung 29: Sichtenkonsolidierung überlappender Sichten	35
Abbildung 30: Beispilmengen in einem Softwarehaus	37
Abbildung 31: Tabelle eines Softwarehauses	37
Abbildung 32: Relationen mit Fremdschlüsselbeziehung	42
Abbildung 33: Relation Personen - Projekte	52
Abbildung 34: Beziehungen von Normalformen zueinander	57
Abbildung 35: Client/Server System.....	60
Abbildung 36: Löschregeln Softwarehaus	70
Abbildung 37: Ausführung SQL-Statement SELECT.....	82
Abbildung 38: Ergebnis des GROUP BY	85
Abbildung 39: Ergebnis-Schema von Kunde und Auftrag.....	90
Abbildung 40: Ergebnisse verschiedener Joins	93
Abbildung 41: Bestellwesen im Softwarehaus	99
Abbildung 42: Schematischer Ablauf einer Transaktion	101
Abbildung 43: Konkurrierende Zugriff auf eine Datenbank	102
Abbildung 44: Isolationsgrad der Sperrumfänge.....	105
Abbildung 45: Index-sequentielle Speicherung	108
Abbildung 46: Schlüssel und Zeiger im Knoten	109
Abbildung 47: B-Baum	109

Tabellenverzeichnis

Tabelle 1: SQL-Standard	59
Tabelle 2: Auswahl an Datentypen	63
Tabelle 3: Constraint	64

Nutzungshinweis:

**Diese Unterlagen dürfen ausschließlich von Mitgliedern
(das sind Studierende, Bedienstete)
der Dualen Hochschule Baden-Württemberg Stuttgart eingesetzt werden.
Eine Weitergabe an andere Personen oder Institutionen ist untersagt.**

1 Grundlagen und Begriffe

Der Begriff Datenbanken wird heute so inflationär verwendet, dass es als allererstes notwendig ist, sich mit der Begriffsdefinition auseinanderzusetzen.

1.1 Was sind Datenbanken?

Der Begriff „Bank“ im Deutschen ist ein Homonym und hat daher mehrere Bedeutungen.

- Eine (Park-)Bank passt nicht, denn darauf kann ich mich nur setzen.
- Bank für Geldinstitut ist schon besser, da Geld eingezahlt und wieder abgehoben werden kann. Es wird auch sicher verwaltet.
- Bei einer Datenbank bringe ich Daten hinein und hole diese auch wieder heraus und gehe davon aus, dass meine Daten sicher verwaltet werden.

Begriffsklärung:

Definition Datenbank:

Def1: Eine Datenbank ist ein logisch zusammengehörender strukturierter Datenbestand.

Def2: Eine Datenbank ist eine Sammlung von Daten, die von einem Datenbankmanagementsystem verwaltet wird.

In der zweiten Definition (es gibt noch viele) wird von einer Sammlung von Daten gesprochen.

Der englische Begriff *Database* drückt dies im Prinzip genauer aus als der Begriff Datenbank.

Dabei kann es sich bei dieser „Datenbasis“ um eine beliebige Sammlung von Daten handeln. Es kann auch einen Karteikasten, eine Registratur oder eine Excel-Tabelle sein

Definition Datenbankmanagementsystem (DBMS):

Ein Datenbankmanagementsystem bzw. Data Base Management System (DBMS) ist die Systemsoftware eines Datenbanksystems (DBS) und dient zur Verwaltung der Daten (Konsistenz, Abfrage der Daten, Datenschutz, Zugriffsrechte....).

Die Datenbanken und das DBMS bilden zusammen das **Datenbanksystem (DBS)**. Dies ist mit einer Datenbankkommunikationsschnittstelle bzw. Benutzungsschnittstelle mit der „Außenwelt“ verbunden.

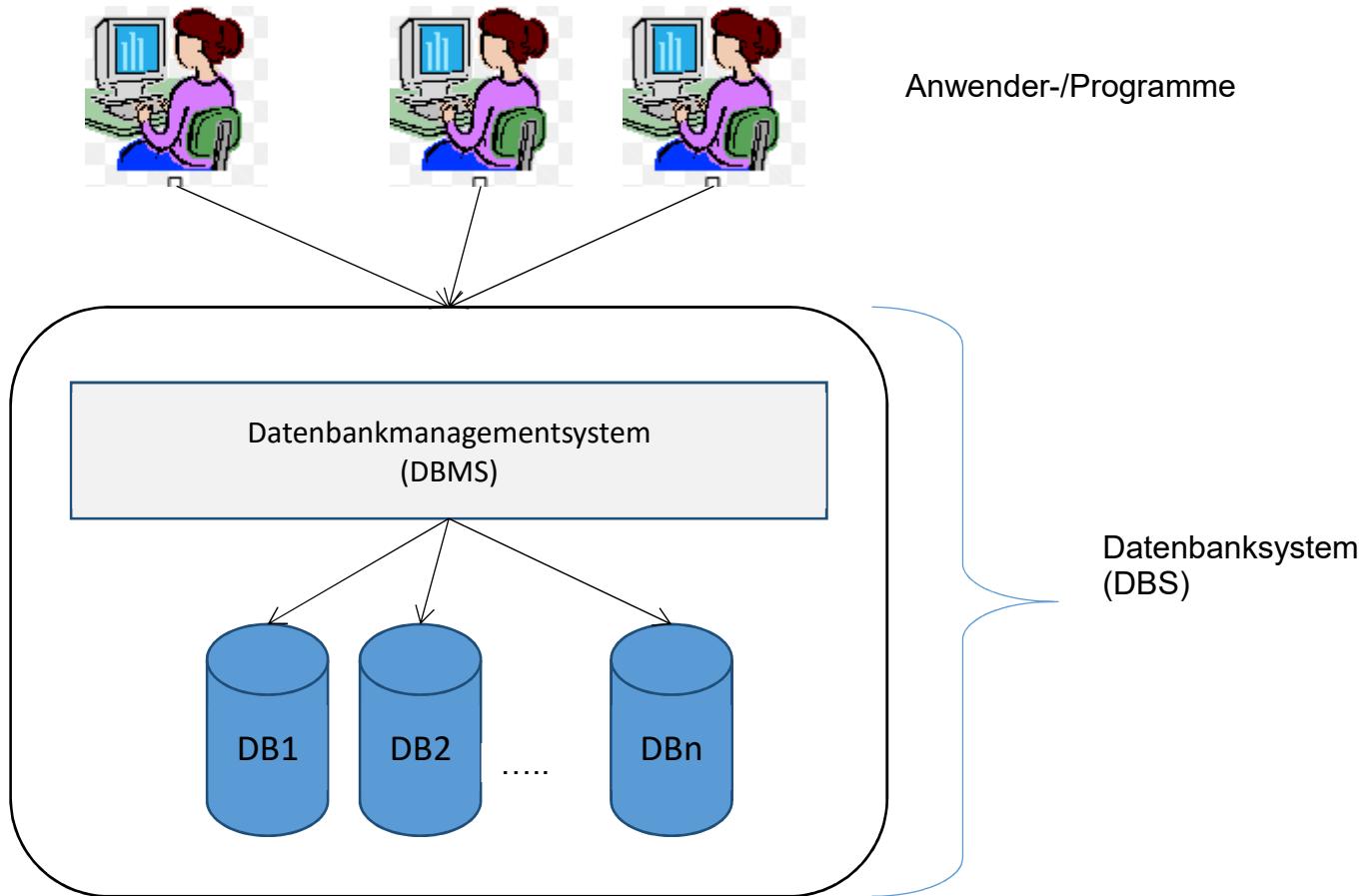


Abbildung 1: Datenbanksystem

1.2 Geschichte

Um zu verstehen, was Datenbanksystems sind und wie das hier als Schwerpunkt zu betrachtende Relationale Datenbankmanagementsystem (RDBMS) entstanden ist, müssen wir uns etwas mit der Geschichte der Informationsverarbeitung und mit der Geschichte des Menschen beschäftigen.

1.2.1 Wie Wissen entsteht

Aus informationstechnischer Sicht entsteht Wissen durch das Sammeln, verdichten, strukturieren, verknüpfen und abspeichern von Daten. Diese werden mit anderen Informationen sinnvoll zu dem was wir Wissen nennen verknüpft. Unser Gehirn ist so in der Lage sehr viel Information aufzunehmen, zu Wissen zu verknüpfen und wieder abzurufen. Es hat aber einen sehr entscheidenden Nachteil. Wissen, welches nicht ständig im Gebrauch ist wird auf der Ebene der Synapsen, sehr schnell wieder abgebaut und geht unwiederbringlich verloren. Was wir aber nicht vergessen wollen oder dürfen, notieren wir uns oder benutzen eine andere Form der Informationsablage wie beispielsweise Käteikärtchen, Notizbücher oder ihre elektronischen Varianten.

1.2.2 Wie entstand die erste Datenverarbeitung?

Unser Gehirn ist im Laufe der Evolution ein sehr effektives Organ geworden, mit dem wir in der Lage sind große Mengen an verschiedenen Informationen, welche über Assoziationen miteinander verknüpft sind, sehr schnell wieder abzurufen. Was unser Gehirn aber nicht kann und dafür wurde

es auch die letzten 50000 Jahre nicht benötigt ist, Daten und Zahlen zu speichern.

In einer Welt in der das Zusammenleben in kleinen Gruppen der Alltag war (bis vor ca. 12000 Jahren), war dies auch keine Anforderung an unser Gehirn.

Vor ca. 12000 Jahren begannen die Menschen dann sesshaft zu werden und die ersten Siedlungen zu bauen. Auch zu dieser Zeit lebten die Menschen in kleinen bäuerlichen Gruppen zusammen und es war daher noch keine Hilfsmittel erforderlich um das Zusammenleben in dieser Gruppengröße zu regeln. Dies war noch problemlos über die Speicherung im Gehirn möglich.

Ca. 3500 Jahren v. Chr. entstanden im fruchtbaren Mesopotamien die ersten größeren Siedlungen mit mehreren hunderten Bewohnern. Nun war es nicht mehr so einfach möglich die Informationen über dieses große Netzwerk in einzelnen Gehirnen zu halten. In dieser Zeit waren es die Sumerer die wahrscheinlich das erste System erfanden mit dem es möglich war, außerhalb des menschlichen Gehirns Informationen abzulegen und weiter zu verarbeiten. Die Sumerer erfanden das erste Datenverarbeitungssystem welches wir „Schrift“ nennen.

Die Schrift ist eine Technik zur Speicherung von Informationen mittels physischen Zeichen. Die Sumerer benutzen dazu zwei Arten von Zeichen. Eine Zeichenart waren die Zahlen und dies war ein Sechtersystem und die zweite Art waren Zeichen für Menschen, Tiere, Waren usw. Diese Informationen wurden auf kleinen Tontäfelchen festgehalten. So war es möglich große Mengen von stupiden Informationen zu speichern, zu der kein menschliches Gehirn in der Lage gewesen wäre. Die Täfelchen dienten aber nur zum Aufzeichnen von Daten aus dem damaligen Geschäftsleben. Mit dieser Schrift war man nicht in der Lage gesprochenes Wort festzuhalten. Daher wird diese Schrift auch als eine partielle Schrift bezeichnet. Andere, heute noch bekannte und sehr wirksame partielle Schriften, sind die mathematische und die Notenschrift.

Diese Informationen auf den Tontäfelchen regelten das geschäftliche Zusammenleben der immer größer werdenden Städte und sogar Reiche. Im Gegensatz zu unserem Gehirn, welches in der Lage ist, die Milliarden gespeicherten Informationen sehr effektiv innerhalb von Sekundenbruchteilen wiederzufinden, war dies auf oftmals tausenden Tontafeln, sehr schwierig bis unmöglich. Um ein funktionierendes Datenverarbeitungssystem zu unterhalten, reicht es leider nicht aus tausende kleiner Tontafeln zu beschriften, sondern dazu war ein nächster Schritt erforderlich. Dies waren Kataloge und Suchsysteme um die gespeicherten Informationen auch effektiv wieder zu finden. Den Ägyptern, Chinesen und anderen Kulturen gelang es äußerst effektive Methoden zur Archivierung, Katalogisierung und zum Wiederfinden von Schriften zu entwickeln. Damit waren die ersten „Datenbanksysteme“ erfunden.

1.2.3 Elektronische Informationsablage

Nun wollen wir einen zeitlichen Sprung machen und befinden uns jetzt in der Mitte des 20igsten Jahrhunderts. Die ersten elektronischen Rechner waren erfunden, dienten aber lediglich zur Verarbeitung von Daten und Informationen, also zum Berechnung. Wie mehrfach in der Geschichte, gab es auch bei der elektronischen Verarbeitung von Daten die Überlegung und dazu verschiedene Projekte, Informationen schnell und effizient zu speichern, zu verknüpfen und wieder abzurufen.

Wie wir oben erfahren haben ist es aus einer unstrukturierten Datenbasis (engl. Database) schwierig, Wissen (verknüpfte Information) zu rekonstruieren. Auch große Mengen an Informationen sind auf diese Weise nicht händelbar (z.B. alle Mitarbeiterdaten einer großen Firma). Die Frage wie beispielsweise – Ich möchte das Durchschnittsgehalt aller Mitarbeiter in unserer Firma wissen – ist nicht, oder nur mit großem Aufwand zu beantworten.

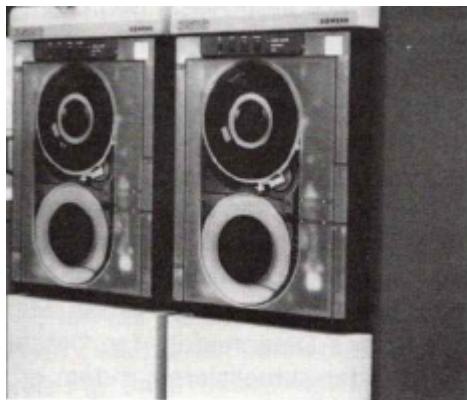


Abbildung 2: Abbildung eines Magnetbandlaufwerks

Zu dieser Zeit entstanden Anforderungen an eine Datenhaltung, die heute noch als Grundlage für DBMS dienen.

- Persistente Datenhaltung
- Verarbeitung großer Datenmengen (Skalierbarkeit)
- Flexibilität
- Sichere Speicherung der Daten
- Zugriff durch mehrere Nutzer
- Zuverlässigkeit
-

1.3 Zeitliche Entwicklung der verschiedenen Datenmodelle

1.3.1 Das hierarchische Datenmodell

Ein hierarchisches Datenmodell ist ein Datenmodell welches die reale Welt in einer Baustruktur abbildet. Der bekannteste Vertreter dieses Datenmodell ist das System IMS (Information Management System) welches in den 1960er Jahren von der Firma IBM entwickelt wurde. IMS ist das älteste kommerzielle Datenbanksystem. IBM hat es einst für die NASA zur Stücklistenverwaltung der Saturn V-Rakete entwickelt. Seit seinem ersten Start im April 1968 ist IMS ständig erweitert und fortentwickelt worden.

Ein typisches Beispiel einer solchen Abbildung ist eine Firma.

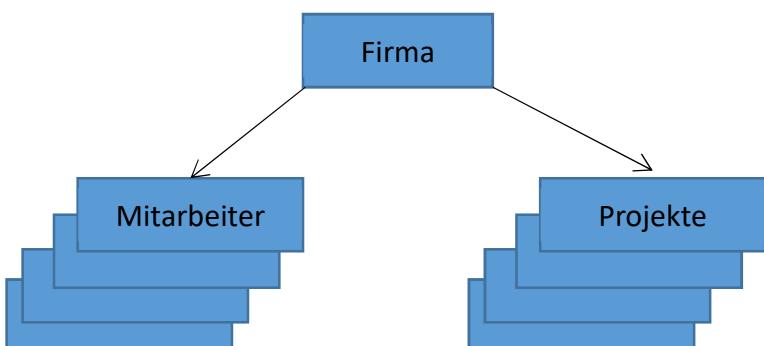


Abbildung 3: Hierarchische Datenmodell

In diesem Modell können nur 1:n Beziehungen modelliert werden. Der Einstieg erfolgt immer über das Wurzelement, welches eindeutig identifiziert werden kann. Ein direkter Zugriff auf den Wert des Datenbestands (z.B. Mitarbeiter) ist nicht möglich.

Da sich die reale Welt aber nicht nur 1:n- Beziehungen zeigt, stößt dieses Modell schnell an seine Grenzen, wenn z.B. die Mitarbeiter einem Projekt zugeordnet werden sollen. Um dies abzubilden, müssten die Mitarbeiter nochmals (redundant) unter die Projekte gehängt werden, oder es wird mit virtuellen Mitarbeitern gearbeitet, welche dann nur als Zeiger unter den Projekten hängen.

Trotzdem sind heute noch viele Systeme mit IMS z.B. bei Banken im Einsatz. Noch heute ist IMS das vermutlich schnellste Datenbanksystem. Dabei ist IMS hinsichtlich CPU- und Speicherbenutzung extrem ressourcenschonend. Eine ganze Reihe moderner Schnittstellen sorgen dafür, dass auch heute übliche web-basierte Anwendungsoberflächen oder Mobile Apps problemlos an das System angebunden werden können, ohne dass vorhandene Daten migriert, oder bestehende Transaktionsprogramme verändert werden müssen.

Eine gewisse Renaissance erlebte die hierarchisch aufgebaute Datenstruktur auch mit XML.

1.3.2 Das Netzwerk Datenmodell

Das hierarchische Datenmodell wurde als proprietäres Datenmodell von der Firma IBM (IMS) entwickelt.

Das Netzwerkdatenmodell wurde auf der Conference on Data System Language (CODASYL) 1971 definiert und wird dadurch oft auch als CODASYL-Datenmodell bezeichnet. Seit 1984 gibt es von ANSI eine Netzwerk-Definitions-Sprache (NDL).

IDMS (Integrated Database Management System) von IBM und UDS (Universal Database System) von Siemens sind kommerzielle Netzwerk-DBMSs.

Im Gegensatz zum HDM welches streng hierarchisch gegliedert ist, kann es im Netzwerk-Datenmodell verschiedene Arten von Beziehungen zwischen den Datensätzen geben.

Soll in einem HDM z.B. eine Teile – Lieferantenbeziehung, bei der in der realen Welt dasselbe Teil von mehreren Lieferanten geliefert wird, abgebildet werden, so geht dies nur mit Redundanzen, oder virtuellen Teilen. Auch in einem einfachen Stammbaum haben in der Regel die Kinder zwei Eltern und nicht nur eine „Vater – Kind“ Beziehung.

Im Netzwerk-Datenmodell ist die Abbildung solcher Beziehungen möglich.

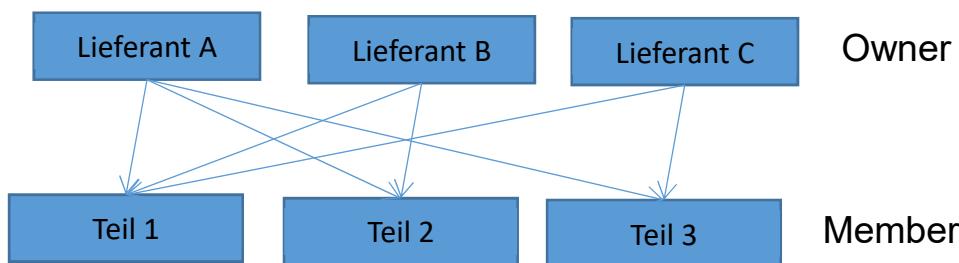


Abbildung 4: Abbildung einer komplexen Beziehung in einem Netzwerk-Datenmodell

Für dieses Modell wurden als ISO-Standard drei Sprachen vorgeschlagen:

- Schema-Datenbeschreibungssprache
- Subschema-Datenbeschreibungssprache
- Datenmanipulationssprache

Diese Sprachen sind komplex und schwer zu erlernen. Der Zugriff erfolgt vor allem aus der Programmiersprache (meistens COBOL) heraus und war daher nur einem kleinen Kreis von Spezialisten vorbehalten. Der modellierte Zustand entsprach auch oft nicht dem Zustand der realen Welt und führte daher auch zu Inkonsistenzen die nicht oder nur schwer erkannt werden konnten.

Auch dieses Datenmodell erlebt eine gewisse Renaissance durch das semantische Web (semantisches Netz, Graphdatenbanken).

1.3.3 Das Relationale Datenmodell

1970 veröffentlichte der Mathematiker E.F. Codd (Forscher bei IBM) seine Idee über relationale Datenbanken. Diese basierten auf dem mathematisch fundierten Grundlagen und dem Modell der Relationen. Sie sollten eine einfache Datenbanksprache besitzen und auch nicht-technischen Benutzern einen Zugang ermöglichen. Die Überwachung der Integrität sollte automatisch und vom DBMS selbst erfolgen.

Es dauert einige Jahre bis Codd bei der IBM die Ressourcen bekam um zu zeigen, dass seine Idee auch praktisch umsetzbar war.

Ende der 1970iger wurde das Projekt erfolgreich abgeschlossen und IBM brachte mit dem „System R“ den ersten Prototyp heraus, welcher in verschiedenen Projekten getestet wurde. Aus System R wurde später DB2. Die Sprache des Systems war SEQUEL, eine eigens für RDBMS entwickelte Abfragesprache, welche später in SQL umbenannt wurde. E.F. Codd erhielt später für seine Leistungen den Turing Award.

Die University of Berkeley veröffentlichte Mitte der 1970iger ihre Version eines RDBMS namens Ingres. Weiter bekannte relationale DBMS wie Oracle, Informix, Sybase und MySQL folgten.

Im relationalen Datenmodell werden die Daten der realen Welt als eine Menge von Relationen abgebildet.

1.4 Die Codd'schen Regeln

Im Jahr 1985 veröffentlicht Codd zwölf Kriterien die zeigen sollten, dass ein DBMS relational, also eine RDBMS ist. Viele Datenbankprodukte, welche sich relational nennen, sind dies bei genauer Überprüfung mit diesen Kriterien leider nicht. Die wichtigsten neun Basisregeln werden anschließend kurz dargestellt. Wir werden im Laufe der Vorlesung immer wieder auf einer der Regeln Bezug nehmen.

Integration:

Alle Daten müssen einheitlich abgelegt werden. D.h. die Daten müssen auf dieselbe Art und Weise und möglichst nicht redundant (redundanzfrei) gespeichert werden. Information, welche in verschiedenem Kontext verwendet wird, darf nur einmal abgelegt werden.

Operationen:

Es muss eine Abfragesprache existieren, mit der Daten gespeichert, geändert und nach Daten gesucht werden kann. Die Operationen sollten einfach und einheitlich sein.

Katalog:

Zu den eigentlichen Daten sind beschreibende Informationen sogenannte Meta-Daten im DBMS abgelegt. Hierbei handelt es sich um alle zusätzlichen Informationen, welche die eigentlichen Daten näher beschreiben.

Beispiel: Es existiert ein Artikeldatensatz und an erster Stelle dieses Datensatzes steht die Artikelnummer und bei diesem Feld handelt es sich um einen Integer Wert. An zweiter Stelle steht die Artikelbezeichnung und dieses Feld ist eine Zeichenkette usw.

Benutzersichten (Views):

Es existiert ein einheitlicher Datenbestand auf den es von verschiedenen Rollen (Benutzer) unterschiedliche Anforderungen gibt. Je nachdem, welche Aufgabe die einzelnen Benutzer haben, dürfen sie auch nur auf bestimmte Daten zugreifen oder diese bearbeiten.

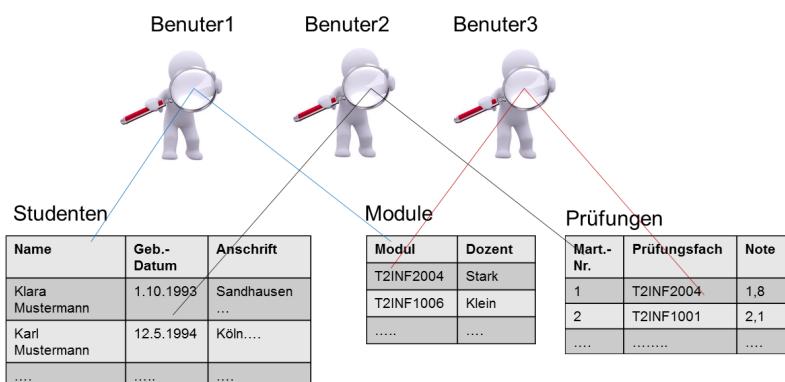
Beispiel:

Abbildung 5: Beispiel für eine Benutzersicht

Konsistenzüberwachung:

Das DBMS führt diese Überwachung automatisch durch. Die Daten werden bestimmten Prüfungen unterzogen damit diese logisch korrekt sind.

Beispiel: Wenn in einem Artikeldatensatz eine Artikelnummer existiert, dann darf diese nicht negativ sein. Oder gibt es in diesem Artikeldatensatz eine Herstellernummer und zu jeder Herstellernummer gibt es in einer anderen Tabelle einen Datensatz, welcher diesen Hersteller näher beschreibt, so darf diese Herstellerdatensatz so lange nicht gelöscht werden, solange noch ein Artikeldatensatz mit diesem Hersteller existiert.

Zugriffskontrolle (Datenschutz):

Es haben nur autorisierte Personen Zugriff auf die Daten. Die Daten sind vor unerlaubtem Zugriff geschützt. Es ist festgelegt, wer lesen, schreiben, ändern oder die Tabelle anlegen und löschen darf.

Transaktion:

Eine Transaktion ist eine Folge von Operationen (Aktionen), die die Datenbank von einem konsistenten Zustand in einen konsistenten, eventuell veränderten, Zustand überführt, wobei das ACID-Prinzip (Atomicity, Consistency, Isolation, Durability) eingehalten werden muss.

Eine Transaktion umfasst eine oder mehrere Datenbankoperationen (Einfügen, Löschen, Modifizieren oder Suchen). Man spricht hier auch von einer logischen Verarbeitungseinheit auf der Datenbank.

Synchronisation:

Ein DBMS erlaubt den parallelen Zugriff für mehrere Benutzer (z.T. sehr viele). Diese „gleichzeitigen“ Zugriffe (lesen, schreiben,...) werden durch das DBMS überwacht. Die Einhaltung der Konsistenz der Daten muss, auch beim Zugriff von Nutzern auf dieselben Daten, immer gewährleistet sein.

Datensicherung:

Tritt ein Systemfehler auf oder „stürzt“ das DBMS ab, so muss eine aktuelle Sicherung existieren welche erlaubt, die Datenbank jederzeit wieder herzustellen. Auch ein Zustand zu einem bestimmten Zeitpunkt muss wiederhergestellt werden können (Recovery).

1.5 Die wichtigsten Begriffe kurz erklärt

Nun wollen wir noch ein paar wichtige Begriffe aus der Datenbankwelt kurz näher beschreiben.

Datenbankmodell:

In den Kapiteln 1.3.1 bis 1.3.3 haben wir den Begriff Datenmodell bereits beim Hierarchischen-, Netzwerk- und Relationalen-Datenbanksystem verwendet, ohne ihn näher zu beschreiben.

Definition:

Bei einem Datenbankmodell handelt es sich um die logische Struktur eines DBMS, welches beschreibt in welcher Form Daten strukturiert, modelliert und abgespeichert werden. Es legt auch fest, welche Operationen zur Modellierung verwendet werden können (Datenbanksprache).

Man könnte auch salopp sagen, es stellt die Infrastruktur bereit, die dazu notwendig ist einen Ausschnitt aus der realen Welt zu modellieren. Auch bei XML handelt es sich im Prinzip um ein Datenbankmodell. Neu hinzugekommen sind in den letzten Jahren die Modelle der sogenannten NoSQL-Datenbanken(Not only SQL) auf die wir in diese Vorlesung aber nicht eingehen werden.

Datenbankschema:

Hier ist es wichtig zwischen Datenbankschema und Datenbankausprägung zu unterscheiden.

Das Datenbankschema beschreibt die Struktur des Datenbestands, also die Metadaten. Wie beispielsweise hängen die Tabellen zusammen, welche Integritätsbedingungen bestehen usw.

Die Datenbankausprägung beschreibt den aktuellen Zustand (welche Daten sind gerade gespeichert) der Datenbasis.

Es gibt zwei Hauptarten von Datenbank-Schemata:

1. Das logische Datenbankschema vermittelt die logischen Einschränkungen, die für die gespeicherten Daten gelten.
2. Das physische Datenbankschema stellt dar, wie die Daten physisch als Dateien und Indizes auf einem Speichersystem gespeichert werden.

Datenbanksprache:

Wie wir bei den Codd'schen Regeln (Kap.1.4) bereits erfahren haben, muss eine DBMS Operationen bereitstellen. Dies erfolgt in der Regel in Form einer Datenbanksprache. Für RDBMS ist diese Datenbanksprache SQL (Structured Query Language). Übersetzt heißt SQL „strukturierte Abfragesprache“ was der Mächtigkeit dieser Sprache aber nicht gerecht wird. SQL besteht aus drei verschiedenen nachfolgend aufgeführten Teilen:

- DDL = Data Definition Language (Anlegen von Tabellen und der zugehörigen Struktur)
- DML = Data Manipulation Language (Einfügen, Ändern und Löschen von Datensätzen)
- DQL = Data Query Language (Abfragen auf den Datenbestand)

1.6 Die 3-Ebenen Architektur

Bereits 1975 entwickelte das amerikanische Normengremium ANSI/SPARC eine drei Schichten Architektur von Datenbanksystemen. Diese besteht aus folgenden drei Ebenen:

- Externe Ebene
- Konzeptionelle Ebene
- Interne Ebene

Dieses Modell beschreibt aber nicht den grundlegenden Aufbau eines Datenbanksystems, sondern es handelt sich hierbei um ein Modell zur Gliederung des Datenbankentwurfes. Hierbei trennt man die drei Ebenen voneinander um diese einzeln und unabhängig voneinander zu entwerfen. Dies hat den Vorteil, dass eine komplexe Datenbankanwendung in drei voneinander unabhängige Teilaufgaben gegliedert werden kann. Man kann sich z.B. mit der konzeptionellen Ebene beschäftigen, ohne die beiden anderen Ebenen ständig im Blickfeld haben zu müssen.

Beginnen wir zunächst mit der konzeptionellen Ebene, welche auch in späteren Kapiteln (siehe Kap. 2- Der konzeptionelle Datenbankentwurf-) noch einen großen Raum einnehmen wird.

Die konzeptionelle Ebene:

In der Regel wird, wenn eine projektgetriebene Anforderung an eine Datenbank abgebildet werden soll, mit der Modellierung dieser Ebene begonnen. Die konzeptionelle Ebene beschreibt, welche Daten gespeichert sind und wie deren Beziehung untereinander ist. Das Ergebnis des Entwurfs sind Tabellen und die Beziehung zwischen den Tabellen. Das Designziel ist eine vollständige und redundanzfreie Darstellung aller zu speichernden Informationen. Diese Ebene wird auch als logische Ebene bezeichnet.

In dieser Vorlesung werden wir uns mit dem Verfahren der Entity-Relationship-Modellierung noch näher beschäftigen.

Die externe Ebene:

Die Nutzergruppen dieser Ebene sind z.T. Endanwender aber auch Anwendungsentwickler, welche ihre speziellen Sichten (Views) auf die logische Eben der Daten zur Verfügung gestellt bekommen. Diese Sichten sind somit die Schnittstelle von der konzeptionellen zur externen Ebene.

Die interne Ebene:

Die interne Ebene wird oft auch als physikalische Ebene bezeichnet. Doch die wirkliche Speicherung der Daten auf der Festplatte wird durch das Betriebssystem gesteuert und daher wird auf dieser Ebene nur die Form der Abspeicherung festgelegt. Beispielsweise die Sortierung nach einem bestimmtem Tabellenfeld, Algorithmen zum Einfügen oder Löschen, oder ob der Zugriff über sogenannte Indexe (z.B. B-Bäume) erfolgt.

Abbildungsfunktion:

Zwischen den einzelnen Ebenen sind Abbildungsfunktionen definiert, welche die Ebenen miteinander verbinden und die Korrespondenz zwischen den Ebenen definieren.

Datenunabhängigkeit:

Weshalb wird nun solch ein Aufwand getrieben, die Ebenen voneinander zu trennen?

Der wichtigste Aspekt dieser ANSI/SPARC-Architektur ist die Datenunabhängigkeit. Diese Datenunabhängigkeit existiert sowohl zwischen der externen und der konzeptionellen Ebene als auch zwischen konzeptioneller und interner Ebene.

- Die logische Datenunabhängigkeit gestattet es, die konzeptuelle Ebene zu ändern ohne die externen Ebenen ebenfalls ändern zu müssen (Transparenz für die Benutzer der externen Ebenen).
- Die physische Datenunabhängigkeit gestattet es, die interne Ebene zu ändern ohne Änderung der konzeptuellen und damit auch externen Ebene vornehmen zu müssen (Transparenz für die Benutzer der konzeptuellen Ebene, insbesondere für Programmierer von externen Ebenen). Die Abbildungen zwischen den Ebenen werden allerdings durch solche Änderungen berührt.

1.7 Die Systemarchitektur

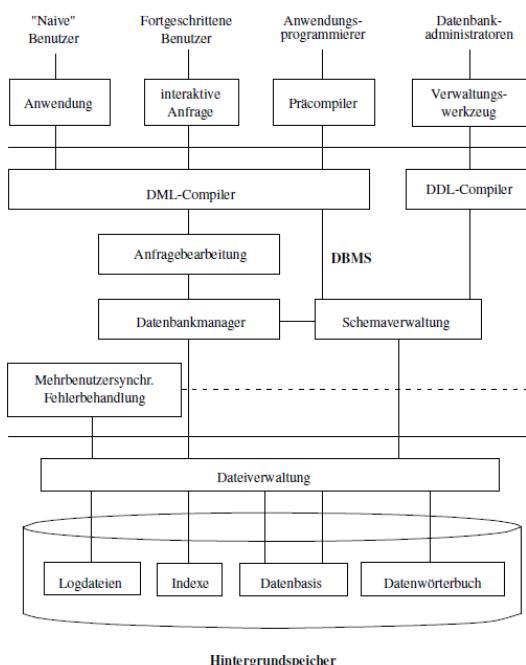


Abbildung 6: Architekturübersicht DBMS

[KeEi15, S.31]

Die Abbildung 6 zeigt einen stark vereinfachten Aufbau eines DBMS.

- Die erste Schicht zeigt die Benutzerschnittstellen, durch die es verschiedenen Benutzergruppen (je nach Anforderung) möglich ist auf das DBMS zuzugreifen.

- Der DML-Compiler (Data Manipulation Language) untersucht die einzelnen Anforderungen und bringt diese in eine Form mit der die Anfrageverwaltung diese weiterverarbeiten kann.
- Die Anfrageverwaltung ist dafür zuständig die einzelne Anfrage so effizient wie möglich mit Hilfe des Datenbankmanager auszuführen.
- Der Datenbankmanager bildet die Schnittstelle zur Dateiverwaltung und führt die Anfragen letztendlich aus.
- Die Mehrbenutzersynchronisation und die Fehlerbehandlung sorgen dafür, dass die Daten vor Zerstörung sicher sind und jederzeit wieder hergestellt werden können.

1.8 Von der Idee zur Datenbank (Entwurfsprozess)

Nur wollen wir uns mit der Frage beschäftigen: Wie gehe ich vor, wenn ich eine Datenbankanwendung entwerfen möchte?

Aus der Vorlesung Projektmanagement kennen Sie klassisches Vorgehensmodell wie beispielsweise das Wasserfallmodell (geht auch bei agilem Vorgehen). Es handelt sich beim Datenbankentwurfsprozess um ein top-down-Vorgehen.

Zuerst erfolgt eine **Anforderungsanalyse** um das Problem bzw. die Anforderungen an die Datenbank zu beschreiben. Dazu ist meist ein Gespräch mit einem Domain-Experten erforderlich. Daraus kann ein Lasten- oder Pflichtenheft oder eine ähnlich gelagerte Beschreibung als Anforderungsspezifikation entstehen. Dieser Schritt legt die Grundlage für die spätere Akzeptanz der Datenbankanwendung, denn was hier vergessen wurde zu erfassen, wird in einem der nächsten Schritte auch nicht modelliert.

Als nächstes erfolgt der **Konzeptionelle Entwurf**, bei dem wir hier mit dem Entity-Relationship-Modell (ERM) als abstrakte Beschreibungssprache arbeiten, auf welches wir im nächsten Kapitel noch genauer eingehen werden. In dieser Stufe des Entwurfsprozesses wird versucht einen Teil der „realen Welt“ mit graphischen Notationen in einer „Miniwelt“ abzubilden. Die Vorgehensweise bei diesem Entwurf ist datenorientiert, dies bedeutet, dass die Daten der Miniwelt und ihre gegenseitigen Beziehungen betrachtet werden.

Der nächste Schritt ist nun der **Logische Entwurf** (Implementierungsentwurf) bei dem man das Schema des Konzeptionellen Entwurfs (welches so nicht implementiert werden kann) in ein Datenbankmodell überführt wird. Dies könnte auch ein Hierarchisches- oder Netzwerk-Datenmodell sein, aber wir werden uns in dieser Vorlesung ausschließlich mit dem Relationalen-Datenmodell beschäftigen.

Als nächstes müssen wir nun die konkreten Datenstrukturen definieren und dieser Abschnitt wird daher auch **Datenbank-Definition** genannt. Dies erfolgt mithilfe der Data-Definition Language (DDL). Die beiden Entwurfsabschnitte des Logischen und der Datenbank-Definition findet man in verschiedenen Publikationen, auch zusammengefasst als den sogenannten Implementierungsentwurf.

Als letzten Schritt müssen wir nun auf der physikalischen Ebene die Datenbank einrichten. Dieser **Physikalische Entwurf** erfolgt unabhängig vom logischen Entwurf und bedarf detaillierter Kenntnisse des jeweiligen DBMS. Dies wird in der Regel vom Datenbankadministrator durchgeführt. Hier wird dann letztendlich festgelegt, wie die Daten gespeichert werden (Pufferstrukturen, Indizes usw.). Die meisten DBMS bieten hierzu verschiedene Möglichkeiten einzugreifen.

Der gesamte Entwurfsprozess sollte konsistent gehalten werden und daher sind Änderungen auf einer unteren Entwurfsebene in den darüber liegenden Ebenen nachzuziehen.

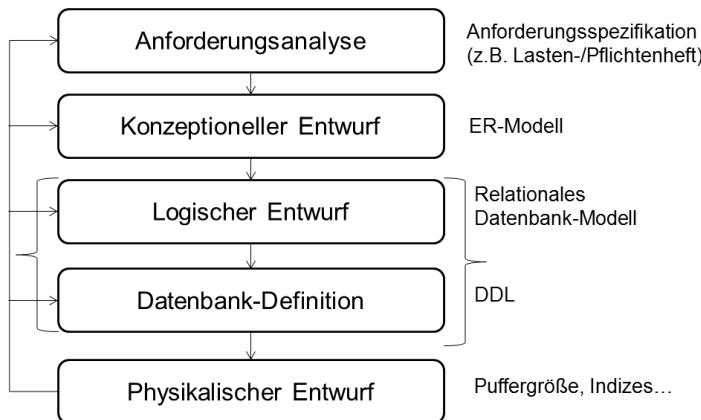


Abbildung 7: Datenbank-Entwurfsprozess

Zum Entwurfsprozess haben wir nun drei unterschiedliche Beschreibungsverfahren erwähnt. Dies sind im Einzelnen:

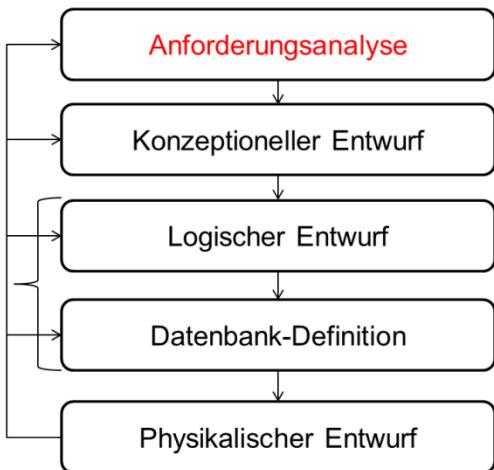
- ER-Modell
- Relationales Modell
- SQL/DDL

Mit diesen Beschreibungsverfahren werden wir uns in den nächsten Kapiteln ausführlich beschäftigen.

2 Der konzeptionelle Datenbankentwurf

2.1 Anforderungsanalyse

Betrachten wir nochmals den Ablauf des gesamten Datenbankentwurfs aus **Abbildung 7**, dann erscheint als Prozessschritt vor dem Konzeptionellen Entwurf noch die Anforderungsanalyse auch Requirements Engineering genannt.



Was versteht man nun unter einer Anforderungsanalyse? Dazu nachfolgend eine Definition:

Requirements Engineering ist ein Begriff des Software Engineerings. Es umfasst das Sammeln, Dokumentieren, Analysieren und Verfolgen von Kundenanforderungen für ein zu erstellendes Softwareprodukt.

Dies bedeutet, dass zuerst eruiert werden muss, was der Kunde für den wir das System (Datenbanksystem) erstellen, für Anforderungen hat. Was sind aber nun Anforderungen?

Hier gibt es viele Definitionen, aber eine der SOPHISTen ([Rupp07], S.13) sei hier stellvertretend aufgeführt:

„Eine Anforderung ist eine Aussage über eine Eigenschaft oder Leistung eines Produktes, eines Prozesses oder der am Prozess beteiligten Personen.“

Bei der Anforderungsanalyse werden die Zielvorgaben, die Anforderungen, das Verbesserungspotential und die Entwicklungsmöglichkeiten untersucht. Das Ergebnis ist ein detaillierter Anforderungskatalog (auch als Lastenheft bezeichnet).

Der Auftrag der Softwareentwicklung ist es nun ein stabiles System zu erstellen, welches die Anforderungen des Kunden abbildet. Dieses Softwaresystem besteht nun aus Funktionen und Daten, welche in einer stabilen Struktur angeordnet sind ([Balz09], S.127)

Zur Dokumentation der Anforderungen können nun verschiedene Techniken zur Anwendung kommen. Diese Techniken dienen dazu das ermittelte Wissen der Stakeholder (Kunden) zu Papier zu bringen. In ([Soph08], S.44) werden drei Gruppen von Techniken unterschieden.

Datenorientierte Techniken zur Beschreibung von statischen Aspekten eines Systems.

Verhaltensorientierte Techniken um das dynamische Verhalten eines Systems zu beschreiben.

Natürlichsprachliche Techniken können keiner der beiden zuvor genannten Techniken zugeordnet werden, oder die Informationen liegen größten Teils in der natürlichen Sprache vor.

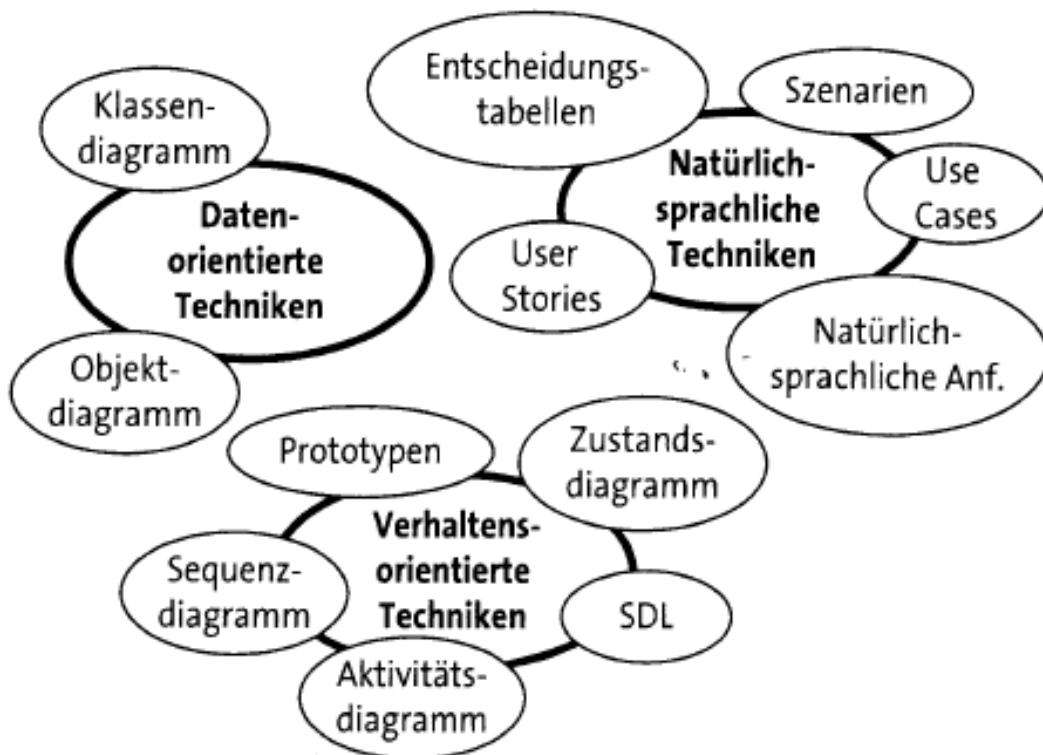


Abbildung 8: Verschiedene Dokumentationstechniken
([Soph08], S.45)

Für diese Vorlesung interessieren uns nur die Daten (Attribute) und der Zusammenhang zwischen Datenelementen, die Datenstrukturen. Hierzu unterstützen uns die Datenorientierteren Techniken. Die Datenelemente können durch statische Beziehungen zu Datenstrukturen verknüpft werden. Hier können beispielsweise baumartige Strukturen mit Hilfe von XML und dem XML-Schemata beschrieben werden. Ein wichtiges und seit vielen Jahren im Einsatz befindliches Modell ist das Entity-Relationship-Modell, kurz ER-Modell. Mit diesem Modell wollen wir uns in diesem Kapitel näher beschäftigen (siehe auch ([Balz09], S.199-214)).

2.2 Zielsetzung

Mit dem konzeptionellen Datenbankentwurf wollen wir nun versuchen einen bestimmten Ausschnitt aus der „realen“ Welt - welche uns unser Kunde beschrieben hat - in einem Datenmodell abzubilden. Diese Entwurfsmethodik „ER-Modell“ wurde 1976 von P. P. Chen entwickelt und ist bis heute im Einsatz. Das Ziel ist es die zu speichernden Daten und ihre Beziehungen zueinander zu analysieren und festzuhalten. Diese Beschreibung ist unabhängig von der Art der späteren Implementierung in einem konkreten Datenbankmodell. Heute werden verbreitet auch OO-Konzepte verwendet, aber das ER-Modell ist vor diesen Konzepten entstanden und stellt einen Spezialfall dar. Das Ergebnis dieses Schrittes im Datenbankentwurfsprozess ist ein vollständiges Modell, eine Art Miniwelt, auf der Basis der Daten unseres zu erstellenden Systems zu beschreiben.

Einige von Ihnen haben in der Vergangenheit vielleicht bereits mit Datenbanken gearbeitet und fragen sich nun, warum benötige ich denn eine Datenmodellierung, wenn ich meine Anforderungen gleich in Tabellen umsetzen kann. Dies ist für kleine Anforderungen, welche sich auch nicht großartig ändern sicher zulässig, aber für große Systeme (welche von IT-Firmen häufig erstellt werden müssen) ist eine Modellierung zwingend erforderlich. In meiner Praxis bei T-Systems gab und gibt es DBMS mit mehreren hundert Tabellen, bei denen es selbst mit einem sauber modellierten ER-Modell schwer fällt, die Übersicht zu behalten.

Wir halten also fest:

Für professionelle Zwecke ist ein konzeptioneller Datenbankentwurf unerlässlich.

2.3 Das Softwarehaus

Um im Folgenden anhand von anschaulichen Beispielen die einzelnen Notationen erklären zu können, werden wir eine Datenbank für ein fiktives Softwarehaus aufbauen.

Der Hintergrund dazu ist der Wunsch der Firmenleitung des Softwarehauses auf Grund des starken Wachstums der Mitarbeiteranzahl und der Projekte, die bisherige Verwaltung in Excel-Tabellen, in ein stabiles Datenbankverwaltungssystem zu überführen.

Unser Softwarehaus führt Projekte für verschiedene Kunden durch und benötigt eine Projektsteuerung und eine Auftragsabwicklung. Das Softwarehaus arbeitet für mehrere Kunden und die Aufträge werden auf Festpreis und nach Aufwand abgerechnet. Auch setzen sich die Aufträge aus mehreren einzelnen Leistungen zusammen.

Das Softwarehaus ist, wie bereits erwähnt, stark expandierend und beschäftigt daher verschiedene MitarbeiterInnen in der Softwareentwicklung und auch in der Administration (Sekretärinnen, Finanzbuchhalter, Führungskräfte). Jeder Mitarbeiter ist einer Abteilung zugeordnet.

Die Projekte werden mit allen anfallenden Daten verwaltet, dies bedeutet, dass sowohl Daten direkt zum Projekt, als auch angefallene Aufwendungen gespeichert werden. Mitarbeiter sind sowohl Projektmitarbeiter als auch Führungskräfte und können auch zeitweise in mehreren Projekten gleichzeitig mitarbeiten. Auch wird in unserem Softwarehaus die Weiterbildung groß geschrieben und daher besuchen die Mitarbeiter verschiedene Kurse um neue Softwaretechniken zu erlernen.

Für unser Beispiel soll dies vorerst genügen, in einer realen Analyse müssten noch mehr Einzelheiten herausgearbeitet werden.

2.4 Die Komponenten des ERM

Der Begriff Entity-Relationship-Modell lässt vermuten, dass die Grundlage bei dieser Art der Modellierung Entitäten (engl. entity) und deren Beziehungen zueinander (engl. Relationship) sind. Was aber ist nun eine Entität?

Die könnten am Beispiel des Softwarehauses Mitarbeiter, Kunden oder Projekte sein.

Unter einer Entität versteht man ein einzelnes, identifizierbares Objekt wie beispielsweise den Mitarbeiter „Müller“ oder den Kunden „Klein“. Für den Entwurf unseres Datenmodells interessiert uns aber nicht das einzelne Individuum, sondern alle Mitarbeiter oder Kunden. Wir sprechen daher von Entitäts-Typen oder Entitäts-Mengen wie Mitarbeiter oder Kunden.

Zu jedem Entitäts-Typ gibt es verschiedene Attribute. Diese dienen dazu die Entitäten oder Beziehungen näher zu charakterisieren.

Nun wollen wir aus unserem Beispiel Softwarehaus die Entitäten identifizieren. Dazu gibt es kein wissenschaftliches Verfahren, sondern hier zählt nur die Erfahrung und Sicht des Modellierers unserer Miniwelt.

Dieses Modell stellt unser Softwarehaus aber nur zum gegenwärtigen Zeitpunkt dar und kann sich jederzeit ändern oder kann erweitert werden. Wenn das Softwarehaus dazu übergeht auch Lizenzen zu verkaufen, so muss dies ohne weiteres in dem Modell abbildbar sein.

Die verschiedenen Entitäts-Typen bildet nun die Grundlage unseres Modells und müssen als Erstes identifiziert werden.

Eine Entität (vereinfacht für Entitäts-Typ) wird in der Notation des ERM als Rechteck gezeichnet welches den Namen enthält. In unserem Modell sind dies Folgende.

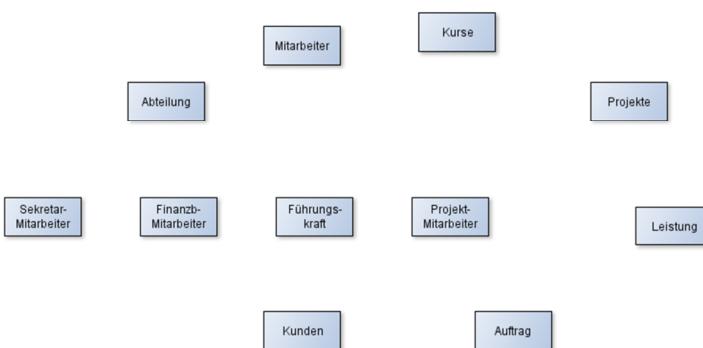


Abbildung 9: Entitäten Softwarehaus

2.5 Die Relationship

Wie schon erwähnt, handelt es sich bei der Relationship um eine Beziehung zwischen zwei Entitäten. Um diese näher zu beschreiben, wollen wir die Entitäten „Mitarbeiter“ und „Projekt“ aus unserem Softwarehaus als einfaches Beispiel verwenden. Mitarbeiter „Müller“ arbeitet im Projekt „Autohaus“ mit. Wie schon bei den Entitäten sind jedoch die einzelnen Beziehungen bei der Modellierung nicht relevant, daher modellieren wir Beziehungs-Typen (-Mengen). Beziehungstypen werden im ER-Diagramm als Raute dargestellt, welche über Linien mit den Entitäten verbunden sind. In die Raute wird der Name der Beziehung geschrieben.

Wenn man erkennen möchte was Subjekt (in unserm Fall Mitarbeiter) und Objekt (Projekt) ist, so ist es in der Regel sinnvoll einen Satz zu bilden.

„Mitarbeiter arbeitet im Projekt“ ergibt durchaus einen Sinn, worauf der Satz „Projekt arbeitet in Mitarbeiter“ sich nicht erschließt.

Die Leserichtung ist in der Regel von links nach rechts, oder von oben nach unten was sich aber nicht immer einhalten lässt. Es ist daher manchmal erforderlich sogenannte Rollen an die Verbindung zu schreiben, um die Eindeutigkeit der Leserichtung festzulegen. Hierauf wollen wir aber nicht näher eingehen.



Abbildung 10: Beziehung Mitarbeiter-Projekt

2.6 Die Attribute und Werte

Die verschiedenen Attribute definieren bestimmte gemeinsame Eigenschaften der Entitäten. Jeder Mitarbeiter hat z.B. einen Vornamen, Nachnamen oder bezieht ein Gehalt. Im Entitäts-Typ Projekt bilden andere Attribute die Eigenschaften ab (Projektnummer, -Name, -Typ...). Die einzelnen Ausprägungen (Gehalt von Herrn Müller) werden als Attributwerte bezeichnet.

Wertebereich

Welchen konkreten Wert nun ein Attribut annehmen kann, wird durch den Wertebereich des Attributs bestimmt. So liegt beispielsweise das Gehalt eines Mitarbeiters in einer bestimmten Grenze und es handelt sich um einen Integerwert. Diese Wertebereiche sind für die einzelnen Attribute oft nicht formal definierbar, aber dass „ABC“ kein Wert eines Vornamens darstellt, braucht sicher nicht begründet werden. Beim Gehalt ist dies schon schwieriger, dass derzeit dieser Wertebereich wahrscheinlich zwischen einem dreistelligen und einem sechsstelligen Wert liegt heißt nicht, dass dies in der Zukunft auch zutreffen muss (Hyperinflation). Trotzdem ist es sinnvoll die Wertebereiche beim Konzeptionellen Entwurf grob festzulegen, da dies sich später bei der Tabellendefinition im Relationalen Entwurf als nützlich erweisen kann.

Welche Attribute sollen nun berücksichtigt werden?

Auf den ersten Blick erscheint diese Frage als unnötig, denn wir wissen doch aus der Anforderungsanalyse genau welche Attribute der Kunde für sein System wünscht. Doch genau hier existiert eine potentielle Fehlerquelle, denn nachträgliche neue Attribute in eine Tabelle einzufügen kann zu Problemen führen.

Wir sollten daher etwas Aufwand auf die Festlegung der Attribute verwenden. Es gibt natürlich Attribute, welche für unser derzeit zu modellierendes Datenmodell nicht relevant sind, wie z.B. die Haarfarbe des Mitarbeiters. Dies trifft aber nur auf das Softwarehaus zu, würden wir eine Datenbank für Haarstudios erstellen, hätte dieses Attribut sicher eine gewisse Wichtigkeit. Schwieriger wird dies schon bei der Anzahl der Kinder oder dem Alter der Kinder, da dies z.B. bei einer Kindergeldauszahlung (die derzeit vielleicht noch nicht stattfindet) wichtig sein könnte.

Attribut im ER-Diagramm

Die Notation für Attribute im ER-Diagramm ist der Kreis in dem der Name des Attributes steht. Dieser ist mit einer Linie mit der Entität verbunden.

Die Wertebereiche sind auf Grund der Übersichtlichkeit nicht aufgeführt, sollten aber wie erwähnt schriftlich festgehalten werden. Wir wollen nun für den Entitäts-Typ „Mitarbeiter“ die relevanten Attribute darstellen.

Pers-Nr, Vorname, Nachname, Geschlecht, Geb-Name, Eintrittsdatum, Skill, Gehalt, Adresse, Alter, Telefon-Nummern

Die Abbildung 11 zeigt die graphische Abbildung der Attribute mit den abgekürzten Namen.

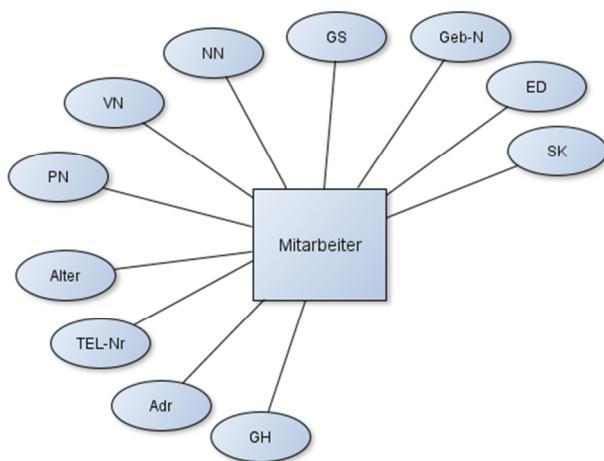
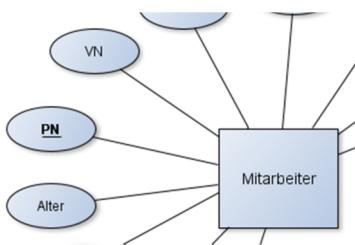


Abbildung 11: Entität mit Attributen

Schlüsselattribute

In unserem Beispiel gibt es verschiedene Arten von Attributen. Die ersten mit dem wir uns beschäftigen möchten sind die sogenannten Schlüsselattribute oder auch Entitätsschlüssel genannt. Dieses Attribut ist ein identifizierendes Attribut, dies bedeutet, dass damit eine Entität (Tupel) exakt identifiziert werden kann, wodurch wir zwei verschiedene Mitarbeiter unterscheiden können.

Wenn wir uns unsere Attribute anschauen so ist man geneigt, die Kombination aus Vorname und Nachname als Schlüssel zu wählen. Dies bedeutet, dass wir zukünftig in unserem Beispiel keinen neuen Mitarbeiter mit dem Namen „Rita Schulze“ mehr einstellen dürfen, was sehr ungeschickt wäre. Auch eine Hinzunahme weitere Attribute zu der Kombination - obwohl dies sehr unwahrscheinlich ist - führt nicht zu einer eindeutigen Unterscheidung. Da die Unterscheidung über die „natürlichen“ Attribute nicht möglich ist, müssen einen sogenannter „künstlicher“ Schlüssel, die Personalnummer (Pers-Nr) einführen. Dies erfolgt immer dann, wenn die Bestimmung eines eindeutigen Schlüssels aus den natürlichen Attributen nicht möglich ist. Dieses Schlüsselattribut stellt keine reale Eigenschaft des Mitarbeiters dar und kann daher eindeutig definiert werden. Im ER-Modell ist dieser Schlüssel nicht so wichtig, aber es muss ein Attribut oder eine Attributkombination gefunden werden, welche die Entitäten unterscheidbar macht. Dieses Attribut wird im Modell dadurch gekennzeichnet, dass der Name unterstrichen wird.

**Abbildung 12:** Entität mit Schlüsselattributen

Wie wählen wir den Schlüssel nun aus?

Da für die spätere Umsetzung in den relationalen Datenbankentwurf der Schlüssel (auch Primär-schlüssel genannt) essentiell wichtig ist, wollen wir uns noch kurz etwas näher mit diesem Thema beschäftigen.

Zuerst müssen alle aus der Entwurfsphase identifizierten natürlichen Attribute auf ihre Tauglichkeit als sogenannte Schlüsselkandidaten hin untersucht werden. Wie bereits erwähnt, kann der Schlüssel auch aus einer Kombination verschiedener Attribute bestehen. Es sollten aber nicht zu viele Attribute kombiniert werden, da auch dies wieder bei der späteren Umsetzung Probleme bereitet.

Zwei Eigenschaften muss ein Schlüsselattribut erfüllen.

1. Es darf sich zukünftig nicht ändern, es muss also zeitinvariant sein.
2. Es muss einen Wert enthalten, darf also nicht NULL sein.

Machen wir dazu ein kleines Beispiel.

Wir haben einen Entitäts-Typ KRAFTFAHRZEUG mit folgenden Attributen:

Entitätsmenge		
Name	Kraftfahrzeuge	
Attribute	Name	Wertebereich
	KENNZEICHEN	FZG-Kennzeichen
	FGST NR	Fahrgestellnummern
	MARKE	KFZ-Hersteller
	LEISTUNG	Kilowatt-Betrag

Nun haben wir auf den ersten Blick mehrere Attribute, welche als Schlüsselkandidaten in Frage kommen und wollen diese auf ihre Tauglichkeit hin untersuchen. Wir nehmen auch an, dass nur zugelassene Fahrzeug eingetragen sind.

Damit können das KENNZEICHEN und die FGST-NR als Schlüsselkandidaten identifiziert werden. MARKE UND LEISTUNG sind nicht eindeutig und eignen sich daher nicht. Aber durch die oben definierte Eigenschaft, dass sich Schlüssel nicht ändern darf, scheidet auch das KENNZEICHEN aus, da sich dieses z.B. bei einem Umzug ändert und daher nicht zeitinvariant ist. Wenn wir die Einschränkung der zugelassenen Fahrzeuge aufheben, so könnte das KENNZEICHEN auch NULL werden und ist somit per Definition auch kein Schlüsselkandidat mehr. So fällt unsere Wahl auf die FGST-NR, welche über die gesamte Lebensdauer eines Fahrzeugs gleich bleibt und weltweit einmalig ist.

Noch ein paar Sätze zu der Definition von künstlichen Schlüsseln. In den meisten Fällen ist es uns nicht möglich aus einem oder mehreren natürlichen Attributen einen Schlüssel zu bilden. Daher ist die Bildung eines künstlichen Schlüssels der Regelfall. Falls man als Designer nun versucht ist bei der Personal- oder einer Rechnungsnummer z.B. das Jahr und eine generierte laufende Nummer zu

verwenden, sollten daraus zwei Attribute gebildet werden aus denen sich dann der Schlüssel zusammensetzt.

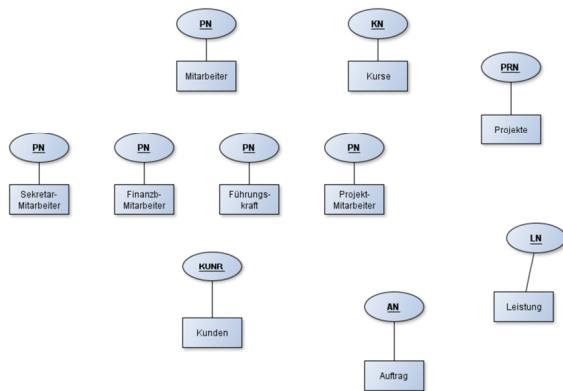


Abbildung 13: Alle Entitäts-Mengen des Softwarehauses mit Schlüsselattributen

2.7 Weitere Attribute

Erinnern wir uns nochmals kurz an die Attribute in Entität „Mitarbeiter“ (*Adresse, Alter, Telefon-Nummern*).

Zuerst betrachten wir das Attribut *Adresse*, welches sich in der Regel aus der Ort, PLZ und Straße zusammensetzt. Hier spricht man von sogenannten zusammengesetzten Attributen, welche aus der Kombination mehrerer Attribute, die inhaltlich zusammengehören, bestehen. Diese sollten in ihre Bestandteile zerlegt und für jedes Teil ein Attribut festgelegt werden.

Das *Alter* ist ein abgeleitetes Attribut welches aus anderen Attributen oder aus Entitäten berechnet wird. Solche Attribute verändern sich ständig und müssen andauern aktuell gehalten werden. Besser ist es das Geburtsdatum des Mitarbeiters als Attribut zu wählen und bei Bedarf das Alter in einem Programm zu berechnen [z.B. *Alter = Jahr (now()) - Geb-Datum*].

Die *Telefon-Nummer* ist das letzte „besondere“ Attribut. Hier handelt es sich um ein Mengenattribut oder mehrwertiges Attribut. Da ein Mitarbeiter in der Regel mehr als eine Telefonnummern hat sehen wir schon am Plural *Telefon-Nummern* (geschäftlich, privat, Handy...) und man auch nicht wirklich weiß, wie viele Nummern letztendlich abgespeichert werden sollen, ist es sinnvoll hierfür eine eigene Entität zu bilden (siehe Kap. 4.1 Anomalien). Die Attribute wären dann Tel-Nr und Art wobei die Telefonnummer der Schlüssel wäre. Die Modellierung könnte wie folgt aussehen.

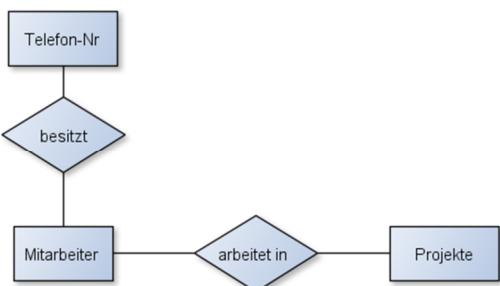


Abbildung 14: Modellierung von Mengenattributen

2.8 Kardinalitäten

Bisher haben wir nur die Beziehung zwischen den Entitäten betrachtet. Beziehungen in einem ER-Diagramm können aber verschieden komplex sein. Wir wollen nun verschiedene Kardinalitäten (Mengenangaben) für Beziehungen betrachten. Es wird auch von der Funktionalität der Beziehung gesprochen (siehe [KeEi15], S41).

1:1 Beziehung

Eine 1:1 Beziehung liegt dann vor, wenn in der betrachteten Beziehung genau eine Entität der ersten Entitätsmenge mit genau einer Entität der zweiten Entitätsmenge verbunden ist. In einem Beispiel aus unserem Softwarehaus bedeutet dies, dass eine Abteilung genau von einem Mitarbeiter geleitet wird. Dies bedeutet aber auch, dass es Abteilung geben kann, die temporär keinen Abteilungsleiter haben und - was sicher einleuchtet - nicht jeder Mitarbeiter eine Abteilung leitet.

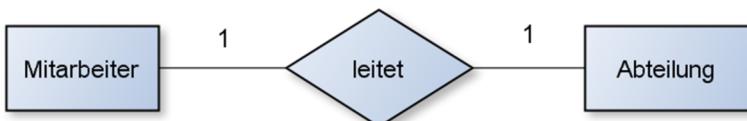


Abbildung 15: 1:1-Beziehung

1:N Beziehung

Nun gibt es in unserem Beispiel auch die Entitäts-Typen „Kunde“ und „Auftrag“. Bei der Modellierung gehen wir davon aus, dass ein Kunde mehrere Aufträge vergibt, aber ein Auftrag nur von genau einem Kunden erteilt wird. Dies bedeutet aber auch wiederum, dass es noch Kunden geben kann, die keinen Auftrag erteilt haben. Auch gibt es vielleicht „interne“ Programmieraufträge, die keinen Kunden haben. Mehrere bedeutet, dass die Anzahl nicht festgelegt ist und daher schreiben wir dies mit einem „N“ (Klein- oder Großschreibung spielt keine Rolle).



Abbildung 16: 1:N-Beziehung

Wie schon erwähnt, ist die Leserichtung im ERM in der Regel von links nach rechts. Die Kardinalitäten für den Kunden (erteilt mehrere Aufträge) werden aber beim Auftrag geschrieben und die Kardinalität, dass ein Auftrag nur maximal von einem Kunden erteilt werden kann, schreiben wir beim Kunden. Die ist die Schreibweise der Chen-Notation, es gibt durchaus noch andere, die wir später kennenlernen werden.

N:M Beziehung

Bei der dritten Komplexitätsstufe betrachten wir nun die Frage wieviel Mitarbeiter arbeiten denn nun konkret in einem Projekt. Also ein Projekt ohne Mitarbeiter ergibt wohl keinen Sinn, daher sollte mindestens ein Mitarbeiter in einem Projekt arbeiten (es muss aber auch möglich sein ein Projekt anzulegen, dem noch keine Mitarbeiter zugeordnet sind). Wieviel können aber nur maximal an einem Projekt arbeiten, da gibt es theoretisch keine Grenze. Auch ist es in einem Softwarehaus

durchaus üblich, dass ein Mitarbeiter in verschiedenen Projekten arbeitet. Hier ist die Anzahl wohl durch die Belastbarkeit des Mitarbeiters bedingt, aber da diese sehr individuell ist (also nicht festgelegt werden kann) schreiben wir hier auch einen Buchstaben für unbestimmt „M“.



Abbildung 17: M:N-Beziehung

An dieser Stelle sollten wir noch kurz auf eine Besonderheit der Beziehungsmenge eingehen. Wenn wir modellieren möchten, wie viele Stunden ein Mitarbeiter an einem Projekt arbeitet, so kann diese Information als Attribut der Beziehung zugeordnet werden, denn beim Projekt selbst macht dies keinen Sinn (siehe Mengenattribute).

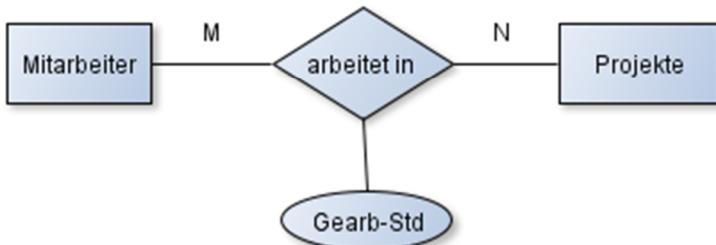


Abbildung 18: Attribute bei Beziehungsmengen

2.9 Weitere Notationen

Es gibt wie erwähnt noch andere Notationsformen, die sich im Wesentlichen dadurch unterscheiden wie die Kardinalitäten dargestellt werden. Zwei sehr gebräuchliche Notationen sind Krähenfußnotation und die Min-Max-Notation, auf die wir noch kurz eingehen werden.

2.9.1 Krähenfußnotation

Bei der Krähenfußnotation oder auch Martin-Notation (nach James Martin, Bachmann und Odell) ist eine etwas einfachere Notation zur Darstellung von ER-Diagrammen. Sie wird so genannt, da die N-Beziehung wie eine Art Krähenfuß im Diagramm dargestellt wird. Die Raute für die Darstellung der Beziehung entfällt und der Text wird auf die Verbindung geschrieben.

Chen-Notation	Krähenfuß-Notation	
1:1	+	- + genau ein
1:N	+	- 0+ kein oder ein
M:N	+	

Abbildung 19: Gegenüberstellung Chen-/Krähenfuß-Notation

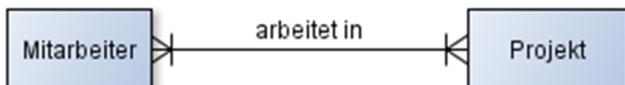


Abbildung 20: Modellierung mit Krähenfußnotion

2.9.2 Die Min-Max-Notation

Nun wollen wir uns noch der letzten in der Praxis wichtigen Notation der (min,max)-Notation (es gibt noch einige mehr) zuwenden. In der Chen-Notation wird nicht präzise ausgesagt, wieviel Tupel letztendlich an der Beziehung beteiligt sein können. Wenn es sich um mehr wie eine Entität handelt sprechen wir von vielen und verwenden die Bezeichnung N. Es kann aber manchmal erforderliche sein, eine genaue Grenze anzugeben und dazu dient diese Notation.

Nehmen wir an, dass unser Softwarehaus auch Inhouse-Kurse für seine Mitarbeiter veranstaltet. Diese kommen aber nur dann zustande, wenn sich mindestens fünf Mitarbeiter anmelden (ist ansonsten zu teuer) und auch die maximale Teilnehmerzahl ist begrenzt. Dies wäre in der Chen-Notation folgendermaßen zu modellieren.

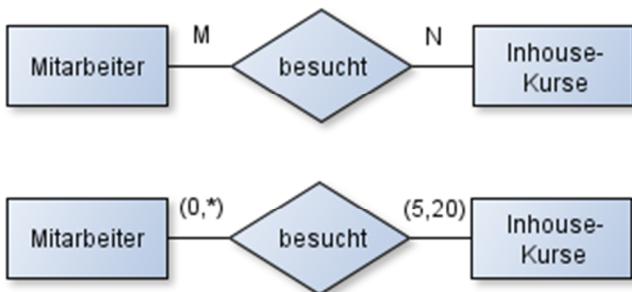


Abbildung 21: (min,max)-Notation vs. Chen

In der Chen-Notation sagt man, dass ein Mitarbeiter an beliebig vielen Kursen teilnehmen kann und ein Kurs auch von beliebig vielen Mitarbeitern besucht werden kann, was in der Praxis allerdings nicht der Fall sein wird. Zur Modellierung reicht dies aber im Allgemeinen aus.

Wenn wir uns die (min,max)-Notation betrachten fällt auf, dass die Schreibweise von Chen abweicht. Da hier konkret die maximale Anzahl der teilnehmenden Entitäten angegeben wird, schreibt man die Notation auf die Seite dieser Entität. Also ein Kurs muss mindestens von fünf Mitarbeitern besucht werden und es können maximal 20 Mitarbeiter teilnehmen (auf Grund der Räumlichkeiten). Ein Mitarbeiter kann noch keinen Kurs besucht haben, darf aber beliebig viele Kurs (natürlich über die Jahre verteilt) besuchen.

Nun verlassen wir die Beziehungsmengen (Kardinalitäten) und wenden uns noch einigen besonderen Beziehungs-Typen zu.

2.10 Besondere Beziehungen

Mehrstellige Beziehungen

Mit der bisher verwendeten zweistelligen (binären) Beziehung lassen sich nicht alle Arten von Beziehungen aus der realen Welt abbilden. Es gibt daher auch n-stellige Beziehungen. Wir wollen uns hier eine drei-stellige Beziehung anschauen. Die Beziehung besteht aus Projektleiter (ist ein spezieller Mitarbeiter des Softwarehauses den wir gleich beschreiben werden), Kunde und Projekt. Die Beziehung hierzu lautet „betreut“. Ein Projekt ist eindeutig einem Kunden zugeordnet und wird nur von einem Projektleiter betreut (geleitet). Ein Projektleiter betreut also ein Projekt für einen bestimmten Kunden. Diese ternäre Beziehung sieht dann wie folgt aus.

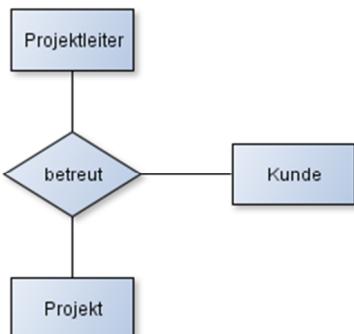


Abbildung 22: Drei-stellige Beziehung

Wie sehen nun die Kardinalitäten in dieser Beziehung aus?

Ein Projektleiter kann mehrere Projekte betreuen und auch mehrere Kunden. Aber ein Projekt wird nur von einem PL betreut und auch ein Kunde hat nur einen PL als Ansprechpartner. Wir sprechen hier auch davon, dass Kunde und Projekt auf PL abbilden (oder eine partielle Funktion bilden). Dies wird dann formal folgendermaßen geschrieben:

betreut: Kunde x Projekt → Projektleiter

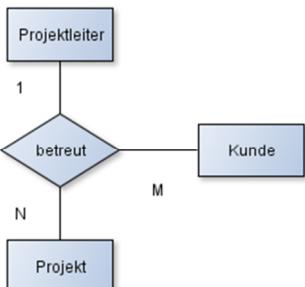


Abbildung 23: Drei-stellige Beziehung mit Kardinalitäten

Überlegen Sie sich selbst dazu, wie für diese Beziehung die Kardinalitäten in der (min,max)-Notation geschrieben werden.

Rekursive Beziehungen

Eine weitere wichtige Beziehung ist die rekursive Beziehung, bei der ein Entitäts-Typ mit sich selbst in Beziehung gesetzt wird. In unserem Softwarehaus beinhaltet der Entitäts-Typ „Mitarbeiter“ alle Mitarbeiter unseres Softwarehauses. Also vom Chef bis zum „kleinen“ Angestellten. Nun ist ein Abteilungsleiter wie gesagt ein Mitarbeiter, aber auch Chef von mehreren anderen Mitarbeitern. Diese Beziehung modelliert man mittels einer Rekursion.

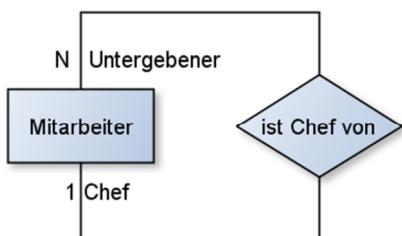


Abbildung 24: Rekursive Beziehung

Dies bedeutet, dass eine Entität aus der Menge „Mitarbeiter“ mit anderen Entitäten der desselben Typs in der Beziehung „Ist Chef von“ steht. Die erstgenannte Entität ist der Chef und die zuletzt genannten sind die ihm unterstellten Mitarbeiter. Man kann für die bessere Darstellung aber auch die Rollen im Modell darstellen.

Spezialisierung und Generalisierung

Mit der Spezialisierung (einer Art Sonderform der hierarchischen Beziehung) wird eine spezielle Teilmenge der Menge gebildet. Dies wird immer dann durchgeführt, wenn sich bei der Modellierung zeigt, dass es Teilmengen gibt, welche besondere Eigenschaften (Attribute) haben. In unserem Beispiel haben wir den Entitäts-Typ „Projekt-Mitarbeiter“ der alle Attribute eines „normalen“ Mitarbeiters besitzt, aber zusätzlich noch weitere Attribute, wie z.B. den *Std-Satz*, den wir dem Kunden in Rechnung stellen und das Attribut *Erfahrung* (in Jahr). Diese Beziehung ist eine sogenannte **Ist-ein-Beziehung** (engl. Is-a) und wird als Sechseck dargestellt.

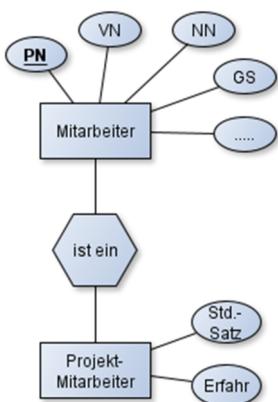


Abbildung 25: Ist-ein- Beziehung

Unser Projekt-Mitarbeiter erbt alle Attribute des übergeordneten verallgemeinerten Entitäts-Typs. Falls in unserem Modell auch externe Mitarbeiter einer Fremdfirma beschäftigt sind, könnten wir eine Spezialisierung in *Interne-Mitarbeiter* und *Externe-Mitarbeiter* durchführen. Hierbei hätten die

externen Mitarbeiter z.B. besondere Attribute wie Tagessatz oder der Namen der Firma aus der sie entliehen sind. Diese beiden Entitäts-Typen wären dann disjunkt, d.h. entweder ist man interner oder externer Mitarbeiter. Spezialisierungen können sich auch teilweise überdecken; d.h. sie müssen nicht disjunkt sein. Würden wir in unserem Modell die *Projektmitarbeiter* und *Interne-Mitarbeiter* als Spezialisierung von Mitarbeitern definieren, so würden sich deren Attribute teilweise überdecken. Ein Projektmitarbeiter erbt aber nicht nur die Attribute des verallgemeinerten Entitäts-Typs, sondern auch seine Beziehungen. Dies leuchtet in unserem Fall auch ein, denn auch ein Projektmitarbeiter kann natürlich einen Kurs besuchen.

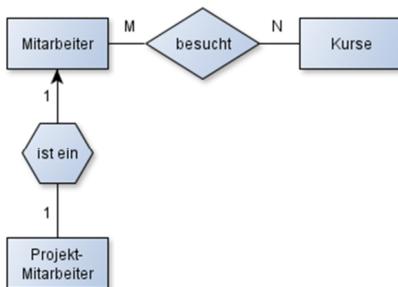


Abbildung 26: Ist-ein- Beziehung mit Beziehung

So kann mit Hilfe des spezialisierten Entitäts-Typs das mehrfache Auftreten gleichbedeutender Attribute und Beziehungen weitgehend vermieden werden.

Um nun erkennen zu können ob eine Spezialisierung sinnvoll ist, so hätten wir bei der Modellierung des Entitäts-Typs *Mitarbeiter* festgestellt, dass die Attribute „Std-Satz“ und „Erfahrung“ bei verschiedenen Mitarbeitern nicht vorhanden sind. Dies deutet darauf hin, dass eine Spezialisierung erforderlich ist. Wir hätten diese Beziehung auch wie in Abbildung 24 dargestellt, als Rekursive Beziehung modellieren können. Die Spezialisierung sollte also nur dann verwendet werden, wenn zusätzliche Attribute beim spezialisierten Entitäts-Typ vorhanden sind.

Auch der umgekehrte Weg ist in der ER-Modellierung durchführbar. Die Generalisierung ist dann sinnvoll, wenn man bei der Modellierung erkennt, dass es verschiedene Entitäts-Typen gibt, welche dieselben Attribute aufweisen. Dann sollte man sich überlegen, ob eine Generalisierung in einen verallgemeinerten Entitäts-Typ sinnvoll ist. In unserem Beispiel könnten wir uns vorstellen, dass der *Kunde* und der *Mitarbeiter* verschiedene gleiche Eigenschaften wie Vorname, Name, Ort,... haben, aber der Kunde hat noch weitere Attribute wie beispielsweise die Rechnungsanschrift. Nun wäre es denkbar (ob sinnvoll müssen wir von Fall zu Fall unterscheiden), dass wir einen generalisierten Entitäts-Typ *Person* bilden, bei dem wir alle gemeinsamen Attribute zusammenziehen.

Aggregation und Partitionierung

Als letzte Form der besonderen Beziehungen wollen wir uns mit der Aggregation/Partitionierung beschäftigen. Diese ist dann sinnvoll, wenn wir erkennen, dass sich eine Entität aus weiteren Entitäten zusammensetzt. Diese haben dann aber nicht zwingend dieselben Eigenschaften. Hierbei werden verschiedene Entitäts-Typen nach oben hin in einem Entitäts-Typ zusammengefasst. Die Beziehung wird als **Teil-von** (engl.is-part-of) bezeichnet. Diese Beziehung findet immer dann Anwendung, wenn ein Entitäts-Typ in weitere Teile zerlegt werden kann (Partitionierung) oder wenn beim umgekehrten Weg Teile zu einem Ganzen zusammengefasst werden können (Aggregation). Da sich in unserem Softwarehaus dazu kein passendes Beispiel finden lässt, wählen wir ein Auto zur Erklärung aus.

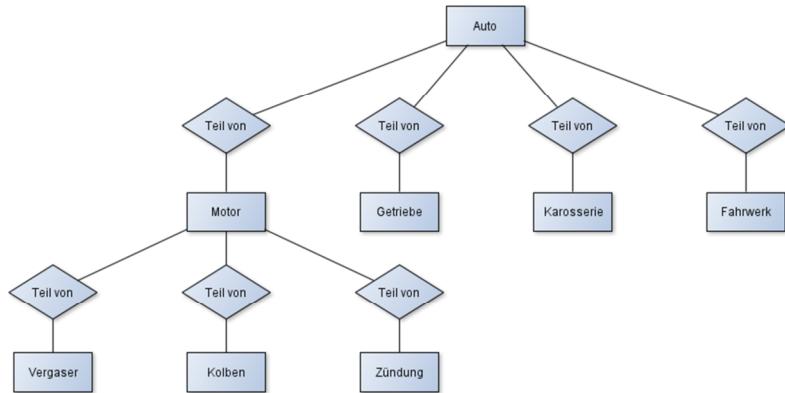


Abbildung 27: Partitionierung am Beispiel Auto

2.11 Schwache Entitäts-Typen

Hier spricht man auch von existenzabhängigen Entitäten, da diese in ihrer Existenz von der „starken“ Entitäts-Typen abhängig sind und ohne diese nicht bestehen. Machen wir dazu wieder ein Beispiel um dies zu verdeutlichen. Im Softwarehaus gibt es Kunden und diese erteilen Aufträge. Zu den Aufträgen gehören bestimmte Leistungen. Hier existiert also eine hierarchische Beziehungskette. Ein Auftrag kann nicht ohne einen Kunden existieren und eine Leistung gibt nur dann einen Sinn, wenn es dazu auch einen Auftrag gibt. Diese schwachen Entitäts-und Beziehungs-Typen werden mit einer doppelten Umrandung und einem Doppelstrich in Richtung der schwachen Entität modelliert. Manchmal wird auch ein Pfeil zur starken Entität hin gezeichnet. Da eine schwache Entität ohne starke nicht existieren kann, hat diese auch keinen eigenen Schlüssel. Zur Identifizierung der schwachen Entität wird eine Kombination des Schlüssels der starken Entität zusammen mit einem Attribut der schwachen Entität verwendet. Z.B. Kundennummer mit Auftragsnummer, oder Auftragsnummer zusammen mit Leistungsnummer.

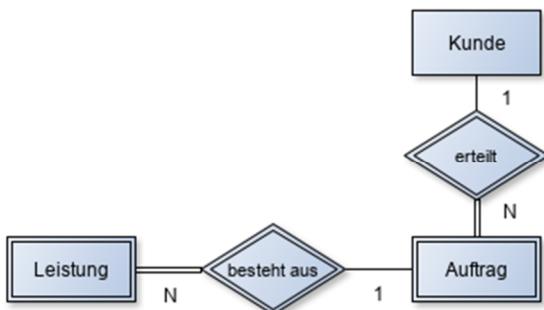


Abbildung 28: Schwache Entitäts-Typen

2.12 Sichtenkonsolidierung

Zum Abschluss des konzeptionellen Datenbankentwurfs wollen wir uns noch kurz mit einer Problematik beschäftigen, die alle Personen (Berater, Architekten...), die beim Kunden unterwegs sind, kennen. Erinnern Sie sich noch an **Abbildung 5**, diese zeigt die verschiedenen Sichten der Benutzer auf die Datenbank. Stellen wir uns nun vor, dass wir, um das Datenmodell unseres Softwarehauses zu modellieren, verschiedene Benutzer befragen. Wir greifen dazu drei heraus.

Eine Mitarbeiterin aus der Personalabteilung, einen Verkäufer und einen Projektleiter. Welche Sicht würden diese Ihnen nun bei der Befragung aufzeigen? Natürlich immer diejenige aus der sie das Softwarehaus betrachten. Bei der Personalerin wäre dies die Mitarbeiterdaten, die Abteilungen und vielleicht noch die Kurse und welche Daten sie benötigt. Die Aufmerksamkeit des Verkäufers liegt auf den Kunden, den Verträgen und den zugehörigen Leistungen (vielleicht noch Projekte). Unser Projektleiter letztendlich sieht sein Projekt und die zugehörigen Leistungsverrechnungen. Diese drei willkürlich ausgewählten Personen schildern Ihnen bei der Befragung ihre individuelle Sicht auf das Datenmodell. Diese Sichten überdecken sich teilweise (nicht disjunkt), unterscheiden sich aber voneinander. Es wäre zu Beginn sicher nicht sinnvoll die einzelnen Sichten gleich als ein großes Schema modellieren zu wollen, denn dies überfordert auch den besten Architekten. Jedoch müssen irgendwann die einzelnen Sichten „zusammengeföhrt“ werden und diesen Vorgang nennt man Sichtenkonsolidierung.

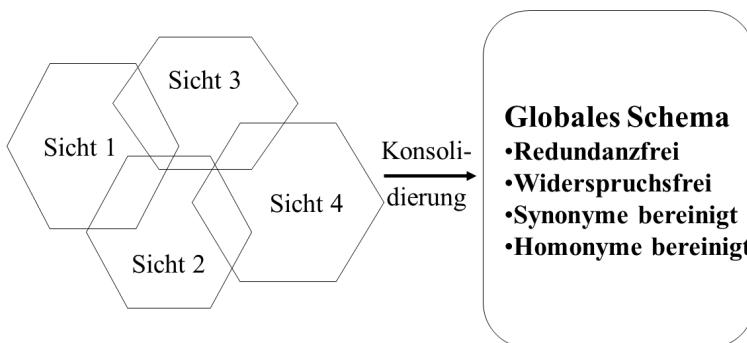


Abbildung 29: Sichtenkonsolidierung überlappender Sichten
[KeEi15, S.56]

Wie in der Abbildung oben gezeigt, muss eine solche Konsolidierung verschiedene Bedingungen erfüllen (redundanzfrei, widerspruchsfrei...). Die Konsolidierung kann nun so erfolgen, dass immer zwei Sichten zusammengeführt werden, um daraus eine gemeinsame Sicht zu erzeugen (z.B. S1 + S2 = S12) danach wird eine weitere Sicht hinzugenommen (S3) usw. Treten Widersprüche auf, was zwangsläufig der Fall sein wird, so müssen diese zusammen mit den Anwendern ausgeräumt werden. Die so modellierte Gesamtsicht muss die Zustimmung aller Anwender finden.

3 Der Relationale Entwurf

Wir haben uns im vorangegangen Kapitel intensiv mit der ER-Modellierung beschäftigt. Wie im Kap. 1.8 bereits erwähnt, ist diese unabhängig von der Umsetzung in ein konkretes Datenmodell. Da wir uns in dieser Vorlesung aber mit Relationalen Datenbanken beschäftigen, werden wir in diesem Kapitel die Umsetzung in dieses Modell behandeln.

Wenn wir uns nochmals kurz an die Abbildung 7: Datenbank-Entwurfsprozess“ erinnern, kommt nun als nächster Prozessschritt der logische Entwurf. Das Relationale Modell ist diesem Prozessschritt zuzuordnen. Nun stellt sich aber die Frage, wie kommen wir vom konzeptionellen Entwurf, zum logischen Entwurf, wie erfolgt hier die Abbildung?

Im Gegensatz zum Hierarchischen- und Netzwerk-Datenmodell ist das Relationale Datenmodell sehr einfach strukturiert. Es gibt im Prinzip nur einfache Tabellen (Relationen) mit ihren Ausprägungen. Die sind:

Zeilen (Tupel), Spalten (Attribute), Zellen (Attributwerte) und die Spaltenüberschriften welche wir als Relationen-Schema bezeichnen. Alles zusammen ist dann unsere Relation mit dem entsprechenden Tabellen-Name (Name der Relation).

Die Verknüpfung erfolgt über entsprechende mengenorientierte Operationen, wie wir im Kap.6.1 noch ausführlich besprechen werden.

3.1 Was versteht man unter einer Relation

3.1.1 Theoretische Betrachtung

In den vorhergehenden Unterkapiteln haben wir die Begriffe Relation/Relationale Datenmodell bereits benutzt, ohne diese näher zu erläutern. Wie bereits erwähnt, war E.F. Codd Mathematiker und daher beschrieb er seine Vorstellung eines neuen Datenbankmodells zuerst in einer sehr abstrakten Sprache, der Relationalen Algebra. Dies war auch der Grund, warum eine Umsetzung seines Modells länger auf sich warten ließ, da sein Modell für die Verantwortlichen zu theoretisch war. Nachfolgend sollen nun die Begriffe erklärt werden.

Definition:

- Eine n-stellige Relation R ist eine Teilmenge des kartesischen Produkts $M_1 \times M_2 \times \dots \times M_n$.
 $R \subseteq \{M_1 \times M_2 \times \dots \times M_n\}$
- Die Wertebereiche M_i heißen Domänen. Domänen sind atomar, d.h. keine zusammengesetzten und oder mengenwertigen Datentypen. Die Anzahl n dieses Bereichs wird als Grad der Relation bezeichnet.
- Ein Element r aus R mit $r = (a_1, a_2, a_3, \dots, a_n)$ mit $a_i \in M_i$ für $i=1, \dots, n$ heißt n-tes Tupel.

Dies ist nun sehr abstrakt und daher soll dieses an einem konkreten Beispiel erläutert werden.

Als Beispiel nehmen wir wieder das Modell des Softwarehauses.

Es beschäftigt verschiedene Mitarbeiter und wir nehmen an, dass zu diesen folgende Informationen gespeichert sind:

Nachname (Nn), Vorname (Vn), Eintrittsjahr (EJ), Geschlecht (G) und der Projektskill (Sk). Es sind sicher noch mehr Informationen gespeichert, die im Augenblick jedoch nicht relevant sind.

M1	M2	M3	M4	M5
Nachname(Nn)	Vorname(Vn)	Eintrittsjahr(EJ)	Geschlecht(G)	Skill(Sk)
Müller	Hans	2001	m	PR
Schulze	Rita	2007	w	DBA
Maier	Werner	2010	m	Test
Schwarz	Karin	2005	w	PR

Abbildung 30: Beispielmengen in einem Softwarehaus

Nun müssen wir, wie in der Definition beschrieben, aus den Mengen das kartesische Produkt bilden. Dies bedeutet, dass wir jedes Element jeder Menge mit jedem Element der anderen Mengen kombinieren.

Müller, Hans, 2001, m, PR

Müller, Hans, 2001, m, DBA

Müller, Hans, 2001, m, Test

Müller, Hans, 2001, w, PR

.....

Schwarz, Karin, 2005, w, PR

Daraus entstehen $N_n=4 \times V_n=4 \times EJ=4 \times G=2 \times Sk=3 = 384$ mögliche Kombinationen.

Dies bedeutet, das kartesische Produkt unserer fünf Mengen enthält 384 Elemente. Wie weiter oben definiert, ist eine Relation eine Teilmenge dieses kartesischen Produkts. Dies könnte eine leere Menge sein, oder 1 - 384 Elemente enthalten.

Die Bestandteile einer solchen Relation werden nun als Tupel bezeichnet (siehe Definition).

Z.B. r1 = Müller, Hans, 2001, m, Pr

Ein Tupel ist ein Element des kartesischen Produkts und die Menge aller Tupel in Abbildung 30 bilden die Relation.

M1 - M5 bilden nun den Wertebereich oder Domain und damit den Grad der Relation. In unserem Beispiel ist die Relation vom Grad 5.

3.1.2 Von der Relation zur Tabelle

Nun wollen wir diese sehr abstrakte Relation etwas konkretisieren und auf eine uns sehr bekannte Form, nämlich eine Tabelle, abbilden.

Mitarbeiter_Softwarehaus									Nr.3
Pers-Nr	Vorname	Nachname	Geschlecht	Geb-Name	Eintritts-Datum	Skill	Gehalt	Nr.5	
1	Hans	Müller	m	NULL	1.7.2001	PR	3200		
2	Rita	Schulze	w	NULL	1.11.2007	DBA	3800		
3	Werner	Maier	m	NULL	1.1.2010	Test	3000		
4	Karin	Schwarz	w	Klein	1.3.2005	PR	3400		

Nr.1

Nr.2

Nr.4

Abbildung 31: Tabelle eines Softwarehauses

Aus Kap. 3.1.1 kennen wir das Tupel und dies entspricht einer Zeile in dieser Tabelle (Nr. 1). Die Menge aller Tupel in unserem Beispiel ist dann die gesamte Tabelle und dies ist unsere Relation (Nr.2). Eine Spalte in der Tabelle entspricht einem Attribut (Nr.3) und eine einzelne Zelle ist dann ein Attributwert (Nr. 4).

Der Tabellenname „Mitarbeiter Softwarehaus“ ist der Name der Relation.

Die Spaltenüberschriften bezeichnen wir als Relationen-Schema (Nr5). Der Grad der Relation entspricht dann in unserer Tabelle der Anzahl der Spalten und daher ist der Grad 8.

Zwei Besonderheiten weist unsere Tabelle noch auf, die wir kurz erläutern wollen.

- Als Erstes die Spalte „Pers-Nr.“ Diese ist eindeutig über alle Mitarbeiter und identifiziert somit unser Tupel exakt. Es darf also keine Pers-Nr. zweimal geben. Diese Spalte wird daher auch als Schlüsselattribut bezeichnet. Kein anderes Attribut unserer Beispieltabelle ist eindeutig und daher sind die anderen Attribute nicht als Schüsselwerte geeignet. Auch die Kombination mehrerer Attribute oder in unserem Beispiel aller Attribute wäre bei einem großen Softwarehaus nicht eindeutig und daher nicht geeignet.
- Die zweite Besonderheit stellt das Attribut „Geb-Name“ dar, in dieser Spalte steht ein sogenannter NULL-Wert, welcher aussagt, dass der Attributwert (Spaltenwert) nicht definiert ist (es steht weder „0“ noch etwas anderes drin). Dies bedeutet, es gibt bei Hans Müller keinen Wert und auch bei Rita Schulz (da sie noch nicht verheiratet ist) ist der Wert nicht definiert.

Wir haben nun die Gemeinsamkeiten zwischen Relationen und Tabellen aufgezeigt, es gibt aber auch durchaus Unterschiede.

- In einer Tabelle können theoretisch auch zwei identische Zeilen auftreten, in einer Relation ist dies nicht möglich. Dies ergibt sich aus der Definition, dass eine Relation eine Teilmenge des kartesischen Produkts der Ausgangsmengen ist. Wenn ein Tupel mehrfach vorkommt, dann könnte die Teilmenge größer als das kartesische Produkt sein und dies entspricht nicht der Definition von Mengen.
- Bei einer Menge gibt es keine Reihenfolge der Tupel, dies ist völlig zufällig (nicht deterministisch).
- Auch bei einer Relation ist die Reihenfolge der Attribute nicht vorbestimmt. Bei einer Tabelle können Zeilen sowie Spalten aber eine Reihenfolge haben (siehe Excel-Tabelle)

Die Attributwerte müssen elementar oder atomar sein, d.h. jeder Attributwert darf nur einen Wert enthalten. Z.B. kann beim Attribut Skill nicht PR und DBA gleichzeitig stehen, obwohl in der Realität eine Person sowohl Programmierer als auch Tester sein kann.

3.1.3 Domänen, Schema und Instanzen

In Kap. 3.1.1 haben wir die Definition des Wertebereichs eingeführt.

Hier haben wir beschrieben: „*Die Wertebereiche M_i heißen Domänen. Domänen sind atomar, d.h. keine zusammengesetzten und/oder mengenwertigen Datentypen*“

Gültig Domänen (Wertbereich) sind Datentypen wie Zahlen, Zeichenketten, Datum usw. Somit definieren wir:

$$R \subseteq D_1 \times D_2 \times \dots \times D_n$$

Beispiel aus der Tabelle Mitarbeiter:

Mitarbeiter ⊆ string x string x integer x string x string

Wie uns das Beispiel zeigt, können die Domänen auch mehrfach vorkommen.

Man unterscheidet bei einer Relation R zwei Bestandteile:

- Einer **Instanz** R: eine Tabelle mit Zeilen und Spalten; der aktuelle Inhalt der Relation (auch Ausprägung genannt)
- Einem **Schema [R]**: spezifiziert den Namen der Relation und die Namen und Datentypen der Spalten; legt die Struktur der Relation fest

Die Relation „R“ ist somit eine konkrete Ausprägung oder die Instanz des Kreuzprodukts. Es handelt sich somit um eine konkrete Menge von Tupel.

Die Menge aller Attribute einer Relation (von n Domänen) ist dann das Schema der Relation.

Die Ausprägung und das Schema werden aber in der Praxis meistens nicht unterschieden und daher mit demselben Namen wie z.B. Mitarbeiter bezeichnet.

Um nun die Domäne des Attributs „A_i“ zu erhalten definieren wir die Funktion

D_i = dom(A_i), somit ist die Relation R

R ⊆ dom(A₁) x dom(A₂) x ... x dom(a_n)

Das Relationenschema wird dann folgendermaßen definiert:

Mitarbeiter : {[Nachname: string, Vorname: string, Eintrittsjahr: integer...]}

(korrekt [Mitarbeiter] : {[Nachname: string, Vorname: string, Eintrittsjahr: integer...]})

Hierbei wird in den eckigen Klammern [...] angegeben, wie die Tupel aufgebaut sind, d.h. welche Attribute vorhanden sind und welchen Wertebereich sie haben. Die geschweiften Klammern { ... } sollen ausdrücken, dass es sich bei einer Relationenausprägung um eine Menge von Tupel handelt. Zur Vereinfachung wird der Wertebereich auch manchmal weggelassen: Die Ausprägung entspricht dem aktuellen Zustand einer Tabelle wie beispielsweise Mitarbeiter:

Hans	Müller	m	NULL	1.7.2001	PR	3200
Rita	Schulze	w	NULL	1.11.2007	DBA	3800
....

Wie im ER-Modell, benötigen wir auch bei der Relation einen Schlüssel (Primärschlüssel). Dieser wird aus der Menge der Attribute festgelegt und es können auch hier mehrere Attribute zu einem Primärschlüssel zusammengefasst werden. Wir kennzeichnen diesen durch unterstreichen des Schlüsselattributs.

Mitarbeiter: {[Pers-Nr: integer, Nachname: string, Vorname: string, Eintrittsjahr: integer...]}

Mit diesem Schlüssel muss eine Tupel aus der Relation eindeutig identifizierbar sein.

3.2 Umsetzung ER- auf Relationales-Modell

Wie kommen wir nun vom ER-Modell zum Relationalen Modell?

Erinnern wir uns nochmals welche Elemente wir im ERM hatten. Dies waren die Entitäts-Typen und die Beziehungen-Typen, sowie die Attribute. Im Relationalen Modell gibt es aber nur die Rela-

tionen und dies bedeutet, dass wir alle Elemente in Relationen überführen müssen. Wir wollen diese Umsetzung wieder an unserem Beispiel Softwarehaus durchspielen.

3.2.1 Überführung von Entitäts-Typen in Relationen

Diese Umsetzung ist recht einfach, denn es werden alle Entitäten-Typen aus dem ER-Modell in jeweils eine Relation überführt. Eine Ausnahme bildet die Spezialisierung, welche wir später behandeln werden. Für unser Softwarehaus müssen wir für alle in Abbildung 9 dargestellten Entitäts-Typen eine Relation bilden. Als Beispiel wollen wir einen Ausschnitt daraus aufzeigen:

Mitarbeiter: {[Pers-Nr: integer, Nachname: string, Vorname: string, Eintrittsjahr: integer, ...]}

Kurs: {[Kurs-Nr: integer, Kurs-Bez: string, ...]}

Kunden: {[Kunden-Nr: integer, Nachname: string, Vorname: string, Firma: string, ...]}

Projekt: {[Projekt-Nr: integer, Bezeichnung: string, Gesamt-Std: integer, ...]}

Projektleiter: {[Pers-Nr: integer, Nachname: string, Vorname: string, Status: string, ...]}

Die Schlüsselattribute aus der ER-Modellierung werden zum Primärschlüssel der Relation welche in der Darstellung unterstrichen sind.

3.2.2 Umsetzung schwacher Entitäts-Typen

Bei der Umsetzung eines schwachen Entitäts-Typs ist der Primärschlüssel nicht nur der Schlüssel der schwachen Entität, sondern dieser setzt sich zusammen aus dem Schlüssel des starken und dem des schwachen Entitäts-Typs. Betrachten wir dazu das Beispiel aus Abbildung 28.



Die Primärschlüssel sind hier die Kundennummer und die Auftragsnummern.

Auftrag: {[Kunden-Nr: integer, Auftrag-Nr: integer, Auftrags-Datum: Datum, ...]}

Hinweis: Es kann aber auch eine eindeutige Auftragsnummer als Primärschlüssel vergeben werden und die Existenzabhängigkeit über eine Löscherregel (siehe Kap.: 5.3.4) erzeugt werden. (bei der Entität *Leistung* macht ein eindeutiger Primärschlüssel jedoch nicht unbedingt einen Sinn).

3.2.3 Umsetzung von Beziehungs-Typen in Relationen

Bei den Beziehungs-Typen ist die schon etwas komplizierter, denn wie wir uns erinnern, gab es im ERM viele Beziehungs-Typen mit unterschiedlichen Kardinalitäten.

Auch zwischen Relationen können Beziehungen auftreten, diese dürfen aber nicht mit denen aus dem ERM gleichgesetzt werden. Im relationalen Datenmodell können an einer Beziehung maximal zwei Relationen beteiligt sein. Außerdem können im relationalen Modell Beziehungen keine eigenen Attribute besitzen, dies muss in anderer Form abgebildet werden.

Beginnen wir mit einer einfachen binären Beziehung und erinnern uns nochmals an ein Beispiel im Softwarehaus.



Aus dieser Beziehung machen wir die Relation „leitet“ und verwenden als Attribute die beiden Schlüssel der beteiligten Entity-Typen.

leitet: {[_Pers-Nr: integer, Projekt-Nr: integer]}

Die Schlüsselattribute aus den jeweiligen Relationen nennt man *Fremdschlüssel*, da diese dazu dienen, die Tupel aus den jeweiligen Relationen (Entity-Typen) zu identifizieren.

Wir können nun aus allen Beziehungen in unserem Modell Relationen machen und diese dann später anhand ihrer Beziehungen verfeinern.

Wie sieht aber nun die konkrete Umsetzung für die verschiedenen Beziehungen aus?

1:N-Beziehung

In unserem Beispiel kann ein Projektleiter verschiedene Projekte leiten, aber ein Projekt kann nur von einem PL geleitet werden.



Wir haben für diese Beziehung damit drei Relationen.

Projektleiter: {[_Pers-Nr: integer, Nachname: string, Vorname: string, Status: string, ...]}
Projekt: {[Projekt-Nr: integer, Bezeichnung: string, Gesamt-Std: integer, ...]}
leitet: {[_Pers-Nr: integer, Projekt-Nr: integer]}

Welches ist nun der Schlüssel in der Relation „leiten“? Hier sagt die Beziehung aus, dass jedes Projekt genau durch einen Projektleiter bestimmt (geleitet) wird. D.h. durch die Projekt-Nummer kann der Projektleiter direkt bestimmt werden (Funktion: *leitet: Projekt → Projektleiter*). Das Projekt (Projekt-Nr) ist dadurch in dieser Relation der Schlüssel, da er ausreicht (kleinster möglicher Schlüssel) um den Projektleiter zu identifizieren.

leitet: {[_Pers-Nr: integer, Projekt-Nr: integer]}

Nun gilt die Regel, dass für 1:1, 1:N und N:1 Beziehungen Relationen mit gleichem Schlüssel zusammengefasst werden können.

Dadurch können die beiden Relationen *Projekt* und *leitet* zusammengefasst werden und die Personalnummer (Pers-Nr) wird als Fremdschlüssel in die Relation *Projekt* mit aufgenommen.

Projektleiter: {[_Pers-Nr: integer, Nachname: string, Vorname: string, Status: string, ...]}
Projekt: {[Projekt-Nr: integer, Bezeichnung: string, Gesamt-Std: integer, geleitet_von...]}

Wenn Sie sich jetzt fragen, warum das Attribut nicht *Pers-Nr* genannt wurde, so ist der einzige Grund, dass dann die im ER-Modell aufgestellte Beziehung verloren geht. Mit dem Namen *geleitet_von* spiegelt man wieder, welche Beziehung wir hier abgebildet haben.

Das nun neu in der Relation *Projekt* eingefügte Attribut *geleitet_von* bezeichnen wir als Fremdschlüssel. In diesem Attribut steht ein gültiger Wert aus der Relation *Projektleiter* und dem Attribut *Pers-Nr*. Diese verbindet die einzelnen Tupel miteinander. Da wir wissen, dass es zu jedem Projekt nur einen Projektleiter gibt, können wir dies durch das Attribut *geleitet_von* ausdrücken.

Schauen wir uns nun einmal ein konkretes Beispiel einer Relation an. In der Relation *Projekte* (1. Tabelle) stehen in der Spalte *geleitet_von* die Personalnummern der Projektleiter aus der Mitarbeiter-Tabelle (Tabelle 2).

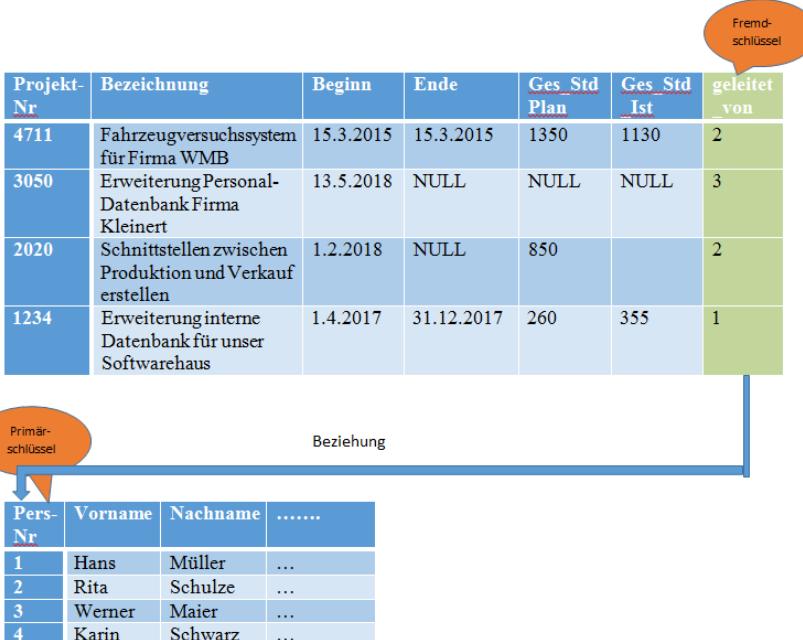
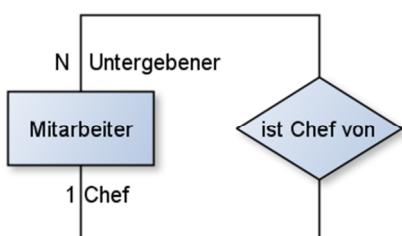


Abbildung 32: Relationen mit Fremdschlüsselbeziehung

Nun kann man sagen, dass diese sehr formale Vorgehensweise dadurch abgekürzt werden kann, dass man den Primärschlüssel aus der Relation der 1-Beziehung als Fremdschlüssel in die Relation der N-Beziehung übernimmt. Anders ausgedrückt können wir - wenn wir etwas Erfahrung im Modellieren haben - gleich zwei Relationsschemas modellieren, ohne den Umweg über eine Beziehungsrelation zu nehmen. Hätte die Beziehung noch eigene Attribute, so würde man diese auch als auf der N-Seite (Tabelle mit Fremdschlüssel) als zusätzliche Attribute mit aufnehmen.

Rekursive 1:N Beziehung



Auch diese Beziehung aus dem ERM bedarf einer besonderen Umsetzung, da der Entitäts-Typ hier mit sich selbst in Beziehung gesetzt ist. Die Beziehungsrelation enthält hier nur die Personalnummer des Mitarbeiters, der Chef der anderen Mitarbeiter ist und kann daher mit der Mitarbeiterrelation zusammengezogen werden. Das neue Attribut *Chef_von* ist nun der Fremdschlüssel für die Tupel der Mitarbeiter.

1:1-Beziehung

In der 1:1-Beziehung hat man mehrere Möglichkeiten zur Umsetzung. Nehmen wir zur Verdeutlichung unser Beispiel aus Abbildung 15.

Hier hatten wir definiert, dass es Abteilung geben kann, die temporär keinen Abteilungsleiter haben und dass nicht jeder Mitarbeiter eine Abteilung leitet. Wir schränken dies nun etwas ein. Wir gehen nun davon aus, dass jede Abteilung von einem Abteilungsleiter geleitet wird (sonst bricht das Chaos

aus) und wenn dieser wechselt, ein Interims-Abteilungsleiter eingesetzt wird. Die Kardinalitäten ändern sich dadurch in der Chen-Notation nicht. Um diese Beziehung dann etwas genauer modellieren zu können, benötigen wir die (min, max)-Notation.



Mitarbeiter: `{[Pers-Nr: integer, Nachname: string, Vorname: string, Eintrittsjahr: integer,...]}`
 Abteilung: `{[Abt_Bez_kurz: string, Abt_Bez_lang: string, Standort: string,...]}`
 leitet: `{[Pers-Nr: integer, Abt_Bez_kurz: string]}`

Wir haben nur zwei Optionen den Primärschlüssel für die Relation *leiten* zu wählen. Im ersten Fall können wir das Attribut *Pers-Nr* zum Primärschlüssel machen und im zweiten Fall die *Abt_Bez_kurz*. Beide Optionen sind bei einer 1:1-Beziehung gleichwertig. Dadurch würde die Vereinfachung der drei Relationen folgendermaßen aussehen.

Option 1:

Mitarbeiter: `{[Pers-Nr: integer, Nachname: string, Vorname: string, Eintrittsjahr: integer, Abt_Bez_kurz: string ...]}`
 Abteilung: `{[Abt_Bez_kurz: string, Abt_Bez_lang: string, Standort: string,...]}`

Hier ist der Fremdschlüssel in der Relation *Mitarbeiter* die Abteilungskurzbezeichnung.

Option 2:

Mitarbeiter: `{[Pers-Nr: integer, Nachname: string, Vorname: string, Eintrittsjahr: integer,...]}`
 Abteilung: `{[Abt_Bez_kurz: string, Abt_Bez_lang: string, Standort: string, Pers-Nr: integer ...]}`

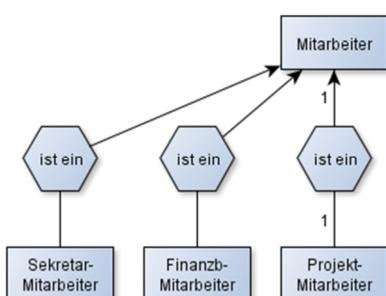
Bei dieser Option ist der Fremdschlüssel in der Relation *Abteilung* die Personalnummer.

Wenn wir uns nun die Beziehung in der (min, max)-Notation anschauen, so fällt auf, dass bei der Relation *Mitarbeiter* eine Null steht, was bedeutet, dass in der Option 1 viele Fremdschlüsseleinträge nicht vorhanden wären und daher mit NULL besetzt wären. Fehlende Fremdschlüssel sollten, wann immer es möglich ist, vermieden werden und daher wählen wir die Option 2.

Betrachten wir uns noch eine häufig vorkommende 1:1-Beziehung, nämlich die Generalisierung (*ist ein*).

Umsetzung der Generalisierung

Für die Generalisierung oder Spezialisierung gibt es im relationalen Modell keine direkte Abbildungsmöglichkeit. Es kommen daher mehrere Möglichkeiten in Betracht, wie wir die nachfolgende Spezialisierung umsetzen können.



1. Die Attribute aller Entitätstypen werden in einer Relation zusammengefasst. Dies bedeutet, dass alle Attribute der spezialisierten Entitäts-Typen (Sekretariats-MA, Finanzb-MA...) und die des verallgemeinerten Entitäts-Typ (hier *Mitarbeiter*) in eine Relation kommen. Da aber die spezialisierten Entitätstypen verschiedene Attribute haben, welche die anderen nicht besitzen, bleiben viele Felder in der zusammengefassten Relation leer (NULL).
2. Für alle Entitäts-Typen wird eine eigene Relation gebildet. Alle an der Spezialisierung beteiligten Relationen besitzen denselben Primärschlüssel. Die Verbindung erfolgt über die in den spezialisierten Relationen eingetragene Personalnummer als Fremdschlüssel. D.h. Personalnummer ist in den spezialisierten Relationen gleich Primär- und Fremdschlüssel.
3. Bei der Dritten Möglichkeit zieht man alle Attribute aus der verallgemeinerten Relation in jede Spezialisierung und erstellt auch für alle eine eigene Relation. Dadurch haben wir alle Informationen in den Spezialisierungen und die Relation *Mitarbeiter* enthält dann keine Tupel mehr, da die Menge aller Tupel in den spezialisierten Relationen, der Menge alle Mitarbeiter entspricht.

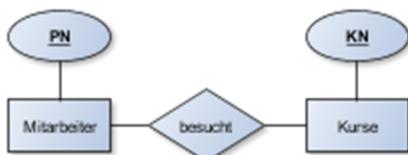
Welche der drei Möglichkeiten wir nur verwenden hängt etwas von der realen Welt ab.

Bei der 1. Möglichkeit besteht der Nachteil, dass wenn die Spezialisierungen viele eigene Attribute haben, dass dann viele Attributwerte NULL sind, was die Relation aufbläht, ohne wirklich Informationen zu erhalten.

Es sollte daher die 2. oder 3. Möglichkeiten gewählt werden, welche auch Vor- und Nachteile haben, auf die wir aber hier nicht näher eingehen werden.

M:N-Beziehung

Auch diese Beziehung wollen wir anhand eines Beispiels erläutern.



Die Relation *besucht* enthält wieder beide Primärschlüsse der beteiligten Entitäts-Typen. Wir nennen diese Relation nun etwas sprechender *besucht_Kurs*.

besucht_Kurs: {[*Pers-Nr*: integer, *Kurs-Nr*: integer]}

Hier ist es nun nicht mehr möglich eine eindeutige Funktion (Abbildung) zu definieren, denn es bestimmt weder ein Mitarbeiter den Kurs, noch ein Kurs wird eindeutig von einem Mitarbeiter bestimmt.

Dies bedeutet, dass beide Schlüsselattribute zusammen den Primärschlüssel der Relation *Besucht_Kurs* darstellen. Wenn die Beziehung noch zusätzliche Attribute hätte wie z.B. einen Termin, so ziehen wir dieses Attribut in die neue Relation, welche dann folgendermaßen aussieht.

<u>Pers-Nr</u>	<u>Kurs-Nr</u>	<u>Termin</u>
2	1312	17.1.2018
4	4711	13.5.2108
1	1312	25.10.2018

Es gibt nun wenig Sinn, wenn wir noch den Termin zum Schlüssel hinzunehmen würden, da dieser durch die beiden Attribute Pers-Nr und Kurs-Nr. eindeutig ist.

Der Schlüssel sollte auch immer minimal gehalten werden.

Die beiden Attribute bilden nun jeweils den Fremdschlüssel für die an der Beziehung beteiligten Relationen *Mitarbeiter* und *Kurs*.

Wir können nun folgende einfache (grobe) Regel für die Umsetzung von Entitäts-Typen und Relationen in das relationale Modell formulieren und diese gelten (vorsichtig geschätzt) für 80% aller ER-Modelle:

1. Alle Entitäten werden Relationen
2. Alle Beziehungen werden Relationen
3. Bei 1:N und N:1 Beziehungs-Relationen können deren Attribute mit der N-Relation zusammengezogen werden. Der Primärschlüssel (der 1-Relation) wird dann zum Fremdschlüssel
4. Bei 1:1 Relationen erfolgt die Zusammenfassung so, dass möglichst wenig NULL-Werte entstehen
5. Aus M:N Beziehungen werden eigenständige (Beziehungs-) Relationen erstellt

3.3 Daten- und Referentielle Integrität

Erinnern Sie sich noch an die Forderung von Codd für ein DBMS (Kapitel 1.4)?

Die Überwachung der Integrität sollte automatisch und vom DBMS selbst erfolgen.“

Was bedeutet dies aber konkret?

Zuerst einmal bedeutet dies, dass die Daten im DBMS konsistent abgespeichert werden müssen.

Hier ist an dieser Stelle nicht die Erhaltung der Daten bei Systemabstürzen, oder bei Mehrbenutzerzugriffen gemeint, sondern hier sind die sogenannten semantischen Integritätsbedingungen gemeint, welche aus der Eigenschaft der modellierten Miniwelt abgeleitet werden können.

Die Datenintegrität ist verletzt, wenn Mehrdeutigkeiten oder widersprüchliche Sachverhalte zutage treten

Die Integritätsbedingungen haben wir schon an verschiedenen Stellen kennengelernt.

1. Der Primärschlüssel darf nicht NULL sein.
2. Es dürfen keine zwei gleichen Attributwerte für den Primärschlüssel existieren.
Ein Tupel muss eindeutig identifizierbar sein.
3. Kardinalitäten der 1:N-Beziehung wurden fest in das RM eingebaut. So leitet ein Projektleiter ein oder mehrere Projekte, aber ein Projekt kann nur von einem Projektleiter geleitet werden. Diese Semantik wurde dadurch modelliert, dass die Personalnummer des Projektleiters als Fremdschlüssel in die Relation *Projekte* aufgenommen wurde. Dies kann nur auf maximal einen Projektleiter verweisen.
4. Für jedes Attribut wurden Wertebereiche (Domäne) festgelegt. So kann die Personalnummer des Mitarbeiters nur Zahlen (integer) und keine andern Zeichen enthalten.

Referentielle Integrität

Für die Konsistenzüberwachung werden mehrere Begriffe verwendet. Engl.: „referential integrity“, deutsch.: „referentielle Integrität“ oder „Beziehungsintegrität“. Wir verwenden hier den Begriff referentielle Integrität.

Hierunter versteht man die Wahrung einer Beziehung zwischen zwei Relationen.

Dies verlangt, dass sich Fremdschlüsselwerte immer nur auf einen Primärschlüsselwerte von existierenden Tupeln beziehen. Die meisten der heutigen relationalen Datenbanksysteme unterstützen die automatische Einhaltung der referentiellen Integrität (muss aber dem DBMS bekannt gegeben werden). Machen wir dazu wieder ein Beispiel.

Fügen wir dazu in die Relation *Projekte* ein neues Tupel ein.

Projekt-Nr	Bezeichnung	Beginn	Ende	Ges_Std_Plan	Ges_Std_Ist	geleitet_von
5505	Fuhrparkverwaltung der Niederlassung Rogge	NULL	NULL	1800	NULL	

Wenn wir für dieses Projekt noch keinen Projektleiter bestimmt haben, so muss der Attributwert in *geleitet_von* auf NULL gesetzt werden. D.h. NULL muss als gültiger Wert zugelassen werden.

Wird nun der Abteilungsleiter bestimmt, so muss dieser als Fremdschlüssel in die Relation eingefügt werden. Würden wir nun gewollt oder versehentlich die Personalnummer „10“ einfügen, so muss uns das DBMS dies abweisen, da es noch keinen Projektleiter mit dieser Personalnummer gibt.

Welche Regeln und Probleme beim Modifizieren und Löschen auftreten können, werden wir in Kap.: 5 -SQL-Teil1- genauer behandeln.

4 Die relationale Entwurfstheorie (Normalisierung)

Das Verfahren der Normalisierung, welches auch von Codd (1972) vorgeschlagen wurde, war ursprünglich ein Entwurfsverfahren, wie die ER-Methode. Als dieses hat es sich wegen des sehr formalen Vorgehens nicht durchgesetzt. Wir verwenden die Normalisierung nachfolgend daher nur als Analyseverfahren um „gute“ Schemata in unserem Modell zu erzeugen. Es wird uns damit möglich, Relationen unabhängig von anderen Relationen zu untersuchen und Mängel im Datenbankentwurf zu erkennen und zu korrigieren. Wir werden uns nachfolgend einzeln die verschiedenen Normalformen (NF) betrachten, von denen in der Praxis die 1NF bis 3NF und die Boyce-Codd-Normalform eine Rolle spielen, die 4.NF und 5.NF sind eher theoretischer Natur und werden daher nur kurz erwähnt.

4.1 Anomalien

Zunächst wollen wir uns einmal ansehen, welche Probleme bei schlecht designten Schemata auftreten können. Nehmen wir dazu die Relationen aus Abbildung 32: Relationen mit Fremdschlüsselbeziehung. Um diese beiden Relationen abzufragen (z.B. welcher Mitarbeiter leitet Projekt „X“), müssen wir die beiden Relationen (Projekte, Projektleiter) miteinander verbinden. Wir könnten aber auch eine neue Relation bilden, die dann folgendermaßen aussehen könnte (einige Attribute wurden auf Grund der Übersichtlichkeit weggelassen).

Proj-Nr	Bezeichnung	Beginn	Pers-Nr	Vorname	Nachname
4711	Fahrzeugversuchssystem für Firma WMB	15.3.2015	2	Rita	Schulze
3050	Erweiterung Personal-Datenbank Firma Kleinert	13.5.2018	3	Werner	Maier
2020	Schnittstellen zwischen Produktion und Verkauf erstellen	1.2.2018	2	Rita	Schulze
1234	Erweiterung interne Datenbank für unser Softwarehaus	1.4.2017	1	Hans	Müller
3091	Elektronische Erfassung der Prüfstandsdaten für Firma WMB	1.9.2018	3	Werner	Maier

Einfügeanomalie

Würden wir in dieser Relation nun einen neuen Projektleiter einfügen wollen, der aber bisher noch keine Projekte leitet, dann müssen wir ein Tupel einfügen, welches z.B. so aussieht.

1010	NULL	NULL	5	Schmidt	Karl
------	------	------	---	---------	------

Um diesen Projektleiter überhaupt einfügen zu können, wurde ein Dummy-Projekt eingefügt, da die Projektnummer der Primärschlüssel ist, kann dieser nicht leer bleiben. Würden wir nun ein neues Projekt anlegen, bei dem der Herr Schmidt der Projektleiter ist, bekommen diese eine neue Projektnummer (1351) und einen vollständigen Datensatz mit Herrn Schmidt als Projektleiter. Unser Projekt 1010 bliebe auf ewig eine „Datenleiche“.

Löschanomalie

Angenommen, das Projekt 1234 würde vom Kunden zurückgezogen werden und wir löschen dann dieses Projekt aus der Relation. Damit würde auch der Projektleiter „Hans Müller“ gelöscht werden, da dieser gerade kein weiteres Projekt leitet. Dies ist sicher nicht im Sinne einer korrekten Verwendung der Relationen.

Änderungsanomalie

Zu einer Änderungsanomalie kommt es dann, wenn z.B. Herr Maier die Firma verlässt und ein neuer Projektleiter benannt wird. Dieser, nennen wir ihn Kunze, soll nun als Ersatz für Herrn Maier bei allen Tupeln eingetragen werden, bei denen seither Maier stand. Dies müssen wir aber bei dieser schlecht designten Relation manuell (im Anwendungsprogramm) durchführen und so kann es möglich sein, dass wir versehentlich einen oder mehrere Tupel vergessen zu ändern. Damit würde sich unsere Relation (Datenbank) in einem inkonsistenten Zustand befinden. Dies darf nach Codd nicht sein.

Um nun die nachfolgenden problematischen Relationsschemata erkennen zu können und die Änderungen und deren Gründe für uns nachvollziehbar sind, benötigen wir noch verschiedene Grundlagen.

Projektion:

Um die nachfolgenden formalen Beschreibungen besser verstehen zu können, müssen wir kurz einen Vorgriff auf Kap. 6.3 durchführen. Hier geht es um den Begriff der „Projektion“ aus der Relationalen Algebra, welche wir im Kap. 6 ausführlich besprechen werden.

Mit der Projektion erhalten wir eine Möglichkeit einzelne Attribute (Spalten) einer Relation auszuwählen. Die Projektion wird mit dem griechischen Buchstaben „Π“ (Pi) bezeichnet.

Allgemeine Definition:

$$R' := \Pi_{[R']} (R),$$

$$[R'] \subseteq [R]$$

Mit ...

$$\Pi_{\text{Vorname}, \text{Nachname}} (M_S)$$

... erhalten wir als Ergebnis nachfolgende Relation:

Vorname	Nachname
Hans	Müller
Rita	Schulze
Werner	Maier
Karin	Schwarz

4.2 Funktionale Abhängigkeit

Wollen wir uns zuerst einmal eine etwas allgemeine Definition anschauen.

Def:

Funktionale Abhängigkeiten (FA; englisch functional dependency, FD) sind Konzepte der relationalen Entwurfstheorie und bilden die Grundlage für die Normalisierung von Relationenschemata. (Wikipedia)

Etwas einfacher ausgedrückt, beschreibt die funktionale Abhängigkeit die Beziehungen zwischen den Attributen einer Relation.

Formal:

Es seien zwei Attributmengen (Teilmengen) X und Y gegeben mit $X, Y \subseteq [R]$
 $X \rightarrow Y$ heißt funktional abhängig (FA) wenn gilt:
es existiert eine Funktion
 $f(x) := y$

für $x \in \Pi_X(R)$, $y \in \Pi_Y(R)$ für alle möglichen Instanzen von R.

Also ganz einfach: Die Werte von X bestimmen die Werte von Y (Y ist funktional abhängig von X).

X heißt Determinante oder linke Seite, Y **Abhängige** oder rechte Seite einer FA [Gebh17, S. 213]

Versuchen wir das anhand eines Beispiels zu erklären.

Wenn wir bei einem Buch die ISBN haben, dann ist davon der Titel, und der Verlag funktional abhängig

$\{ISBN\} \rightarrow \{Titel, Verlag\}$.

Oder ist die Straße und der Ort (für Deutschland) gegeben, dann ist davon funktional die Postleitzahl abhängig

$\{Straße, Ort\} \rightarrow PLZ$

Diese funktionalen Abhängigkeiten müssen für alle Instanzen von (R) gelten und nicht nur für einige konkrete Instanzen, die wir zufällig gerade gefunden haben. Diese Abhängigkeiten beziehen sich auf die Semantik von Attributen und können daher nicht automatisch bestimmt werden, sondern müssen von den Datenbankdesignern festgelegt werden.

Triviale funktionale Abhängigkeit:

Ein Attribut ist immer funktional abhängig:

- von sich selbst
- und von jeder Obermenge von sich selbst

Solche Abhängigkeiten bezeichnet man als trivial.

Formal: $X \rightarrow Y$ heißt trivial, wenn gilt $Y \subseteq X$

(Spezialfall: $X \rightarrow X$)

4.3 Interferenzregel und Armstrong-Axiome

Die funktionalen Abhängigkeiten, welche wir auf das Schema einer Relation bilden können, können mithilfe der Interferenzregeln noch erweitert werden. Diese Regeln sind teilweise für die nachfolgenden Beschreibungen der Normalformen wichtig.

Für ein Relationenschema [R] sind folgende funktionale Abhängigkeiten gegeben.

$$\{PersNr\} \rightarrow \{Name, GebDatum, Geschlecht, AusweisNr, TelNr, Raum, Straße, PLZ, Ort\}$$

$$\{Straße, Ort\} \rightarrow \{PLZ\}$$

$$\{TelNr\} \rightarrow \{Vorname, Nachname, Raum\}$$

$$\{AusweisNr\} \rightarrow \{PersNr\}$$

Aus den diesen können dann weitere FA's abgeleitet werden, die jedoch für die Umsetzung eventuell nicht relevant sind wie z.B.:

$$\{Raum, TelNr\} \rightarrow \{PersNr, Name, GebDatum, Geschlecht, AusweisNr, Straße, PLZ, Ort\}$$

Die Menge aller zu einem Schema [R] definierten funktionalen Abhängigkeiten bezeichnen wir als **F**. Diese Menge lässt sich noch vergrößern bis wir alle möglichen herleitbaren funktionalen Abhängigkeiten einer Relation bestimmt haben und diese nennt man **F⁺**.

Diese wird dann als „Hülle“ der Menge **F** bezeichnet. Die Hülle (**F⁺**) kann durch die Anwendung der Interferenzregeln von Armstrong hergeleitet werden.

Es gilt:

(1)	$Y \in X$	$\Rightarrow X \rightarrow Y$	Reflexivität
(2)	$X \rightarrow Y$	$\Rightarrow XZ \rightarrow YZ$	Verstärkung
(3)	$X \rightarrow Y, Y \rightarrow Z$	$\Rightarrow X \rightarrow Z$	Transitivität
(4)	$X \rightarrow YZ$	$\Rightarrow X \rightarrow Y, X \rightarrow Z$	Zerlegung (Dekomposition)
(5)	$X \rightarrow Y, X \rightarrow Z$	$\Rightarrow X \rightarrow YZ$	Vereinigung
(6)	$X \rightarrow Y, WY \rightarrow Z$	$\Rightarrow WX \rightarrow Z$	Pseudotransitivität

Die ersten drei Regeln reichen bereits für eine vollständige Herleitung von **F⁺** aus. Die Regeln (4-6) sind ganz nützlich für einen komfortableren Herleitungsprozess (dazu sei auf [KeEi15, S.184] verwiesen).

4.4 Schlüssel

Hier wollen wir nochmals zusammenfassen (und etwas erweitern), was wir in den letzten Kapiteln bereits über Schlüssel (-Attribute) erfahren haben (siehe Kap.2.6 „Schlüssel“).

- Ein Schlüsselkandidat ist ein Attribut einer Relation, welches sich potentiell als Schlüssel eignen würde.
- Ein Schlüssel (Primärschlüssel) identifiziert ein Tupel einer Relation eindeutig.
- Ein Schlüssel kann auch aus einer Kombination verschiedener Attribute der Relation bestehen.
- Der Schlüssel sollte nur so viele Attributkombinationen enthalten, wie minimal zur Eindeutigkeit des Schlüssels führen.

Erweitern wir diese Aussagen noch etwas.

Die Relation ist eine Menge von Attributen [R] = {A₁, A₂, ..., A_n}

Superschlüssel

Jede Teilmenge S_i ⊆ [R] für die gilt

$$S_i \rightarrow [R]$$

heißt Superschlüssel von R.

Ein Superschlüssel ist die Menge von Attributen einer Relation, welche ein Tupel in dieser Relation eindeutig identifizieren. Es sagt noch nichts darüber aus, ob es sich um eine minimale Menge von Attributen handelt. Ein Superschlüssel wäre zum Beispiel die Menge aller Attribute einer Relation gemeinsam ($[R] \rightarrow [R]$).

Schlüsselkandidat

Eine minimale Teilmenge der Attribute eines Superschlüssels, welche die Identifizierung der Tupel ermöglicht (Schlüsselkandidaten \subseteq Superschlüssel). In der Regel wird einer dieser Kandidaten als Primärschlüssel ausgewählt. Eine Relation kann mehrere Schlüsselkandidaten (k_j) haben.

Primärschlüssel

Der Primärschlüssel ist dann ein (willkürlich) ausgewählter Schlüsselkandidat, der zur eindeutigen Identifikation der Tupel in der Relation verwendet wird.

Primattribute:

Ein Attribut wird Primattribut oder „prim“ genannt, wenn es in irgendeinem Schlüsselkandidaten von $[R]$ vorkommt.

Formal: Wenn K die Menge aller Schlüsselkandidaten von $[R]$ ist, dann heißen alle Attribute in der Vereinigung über alle j von K_j , prim oder Primattribute.

Alle Attribute in $\bigcup_j K_j$ heißen prim

4.5 Normalformen

Es soll nun geprüft werden, ob die Relationen bestimmten Anforderungen genügen, falls nicht, müssen diese entsprechend transformiert werden, damit sie anschließend diese Normalformen erfüllen. Die Normalformen bauen aufeinander auf, d.h. ist eine Relation z.B. in der 3. Normalform, dann erfüllt sie auch die Bedingungen der 2. und 1. Normalform.

4.5.1 1. Normalform

Die Bedingung, wenn eine Relation nicht dieser Normalform (NF) genügt, haben wir bereits kurz in Kap. 2.7 bei mehrwertigen Attributen (Mengenattributen) besprochen.

Eine Relation erfüllt die erste Normalform, wenn jede Entität (jedes Tupel) für jedes Attribut der Relation nur einen Datenwert besitzt.

Eigentlich könnten wir an dieser Stelle mit dieser 1. Normalform schon enden, denn in der Definition einer Relation ist bereits enthalten, dass eine Relation keine mengenwertigen Attribute enthalten darf (siehe Kap. 3.1.1). Dort haben wir definiert: „Domänen sind atomar, d.h. keine zusammengezetzten und oder mengenwertigen Datentypen.“

Der Vollständigkeit halber wollen wir aber trotzdem nochmals kurz darauf eingehen.

Dazu nehmen wir wieder unser schlecht designtes Schema von oben und nehmen noch das Attribut „Abschlüsse“ darin auf (Geschlecht und Eintrittsdatum haben wir der Übersichtlichkeit wegen weggelassen).

Pers-Nr	Vorname	Nachname	Abschlüsse	Abt_Nr	Abt_Bez	Abt_Leiter
1	Hans	Müller	Dipl. Informatiker (FH) 2000	3	Service	Lehmann
2	Rita	Schulze	Großhandelskauffrau 2000, Dipl. Wirtschaftsinformatik 2004	3	Service	Lehmann
3	Werner	Maier	Fachinformatiker 2000, Kaufmann 2002, Bachelor of Science Informatik 2006	2	Pro-grammieren 1	Langer
4	Karin	Schwarz	Kauffrau für Bürokommunikation 2001, Dipl. Wirtschaftsinformatik 2005	1	Pro-grammieren 2	Hausmann

Hier haben wir wieder dieses typische mengenwertige Attribut, welches wir in eine weite Relation auslagern müssen. Diese Relation könnten wir *Abschlüsse* nennen und die Verbindung mit der Mitarbeiter-Relation erfolgt dann über die Personalnummer als Fremdschlüssel.

4.5.2 2. Normalform

Eine Relation erfüllt die zweite Normalform (2NF), wenn diese sich in der ersten Normalform befindet und alle Nichtschlüsselattribute nur durch den gesamten Primärschlüssel festgelegt werden.

Hier kommt nun wieder der Begriff funktionale Abhängigkeit ins Spiel.

Alle Nichtschlüsselattribute müssen vollfunktional vom gesamten Schlüssel abhängig sein.

Dazu nehmen wir unsere Relation *Projekt* (etwas abgeändert) und verbinden diese mit der Relation *Mitarbeiter* (wir führen einen Join durch).

Pers-Nr	Vorname	Nachname	Projekt-Nr	Bezeichnung	Geleistete_Std
1	Hans	Müller	4711	Fahrzeugversuchssystem für Firma WMB	125
2	Rita	Schulze	3050	Erweiterung Personal-Datenbank Firma Kleinert	206
2	Rita	Schulze	2020	Schnittstellen zwischen Produktion und Verkauf erstellen	110
3	Werner	Maier	1234	Erweiterung interne Datenbank für unser Softwarehaus	154
3	Werner	Maier	2020	Schnittstellen zwischen Produktion und Verkauf erstellen	144
4	Karin	Schwarz	1234	Erweiterung interne Datenbank für unser Softwarehaus	154

Abbildung 33: Relation Personen - Projekte

Projektnummer und Personalnummer bilden in dieser Relation zusammen den Primärschlüssel.

Prüfen wir nun, welche funktionalen Abhängigkeiten es in dieser Relation gibt.

1. $\{Pers_Nr\} \rightarrow \{Vorname, Nachname\}$

-
2. $\{Projekt_Nr\} \rightarrow \{Bezeichnung\}$
 3. $\{Pers_Nr, Projekt_Nr\} \rightarrow \{Geleistet_Std\}$

Es gibt durchaus noch mehr Abhängigkeiten, die uns aber hier nicht weiter interessieren. Die Definition besagt nun, dass sich die Relation in der 1.NF befinden muss, dies ist hier offensichtlich der Fall, da es nur atomare Attribute gibt. Weiterhin dürfen alle Nichtschlüsselattribute und dies sind *Vorname*, *Nachname*, *Bezeichnung*, *Geleistete_Std* nur vom gesamten Primärschlüssel abhängen (vollfunktional abhängig). Wenn wir dazu die funktionalen Abhängigkeiten anschauen, so erfüllt diese Bedingung nur das Nichtschlüsselattribut *Geleistete_Std* (3.). Was aber nun bei einem solchen „schlechten“ Relationschema passieren könnte, wurde bei den Anomalien aufgezeigt.

Um nun die Relation in die 2NF zu bringen, bilden wir zwei neue Relationen und in der ursprünglichen bleiben nur noch die geleisteten Stunden.

Projekt_Stunden: {[Pers-Nr: integer, Projekt-Nr: integer, Geleistet_Std:integer]}

Projekt: {[Projekt-Nr: integer, *Bezeichnung*: string]}

Mitarbeiter: {[Pers_Nr: integer, *Vorname*:string, *Nachname*: string]}

In den zwei neuen Relationen sind die Fremdschlüssel, die Projektnummer und die Personalnummer.

Hinweis: Alle Relationen, bei denen der Primärschlüssel nur aus einem Attribut (nicht zusammengesetzt) besteht, sind automatisch in der 2.NF.

4.5.3 3. Normalform

Eine Relation ist in der 3.Normalform (3NF), wenn diese die 2. Normalform erfüllte und keine transitiven Abhängigkeiten zwischen einem Nichtschlüsselattribut und einem Schlüsselkandidaten bestehen.

Formale Definition:

Ein Relationsschema [R] ist in 3NF, wenn für jede für [R] geltende funktionale Abhängigkeit der Form $X \rightarrow \alpha$, $X \sqsubseteq [R]$ und $\alpha \in [R]$

mindestens eine der drei Bedingungen gilt:

- $\alpha \in X$, (dann ist die funktionale Abhängigkeit trivial)
- Das Attribut α ist in einem Schlüsselkandidaten von [R] enthalten (dies bedeutet α ist „prim“)
- X ist ein Superschlüssel von [R]

Erinnern wir uns nochmals, dass ein Schlüsselkandidat eine minimale Teilmenge der Attribute eines Superschlüssels, welche die Identifizierung der Tupel ermöglicht (Schlüsselkandidaten \subseteq Superschlüssel), ist.

Um dies nun an einem Beispiel zu erläutern, haben wir die Relation (Projekte und Projektleiter), an der wir die Anomalien erläutert haben, hier wieder verwendet.

Proj-Nr	Bezeichnung	Beginn	Pers-Nr	Vorname	Nachname
4711	Fahrzeugversuchssystem für Firma WMB	15.3.2015	2	Rita	Schulze
3050	Erweiterung Personal-Datenbank Firma Kleinert	13.5.2018	3	Werner	Maier
2020	Schnittstellen zwischen Produktion und Verkauf erstellen	1.2.2018	2	Rita	Schulze
1234	Erweiterung interne Datenbank für unser Softwarehaus	1.4.2017	1	Hans	Müller
3091	Elektronische Erfassung der Prüfstandsdaten für Firma WMB	1.9.2018	3	Werner	Maier

Als erstes prüfen wir nun ob sich die Relation in der 1NF befindet. Dies ist gegeben, da es keine mengenwertigen Attribute gibt.

Die Relation ist auch in der 2NF, da die Projektnummer der Primärschlüssel ist und dieser nur aus einem Attribut besteht.

Um nun zu prüfen, ob die Relation in der 3NF ist, müssen wir wieder die funktionalen Abhängigkeiten aufstellen.

$$\{Projekt_Nr\} \rightarrow \{Bezeichnung, Beginn\}$$

$$\{Pers_Nr\} \rightarrow \{Vorname, Nachname\}$$

$$\{Projekt_Nr\} \rightarrow \{Pers_Nr, Vorname, Nachname\}$$

Der Schlüsselkandidat ist in dieser Relation die Projektnummer und von dieser hängen die zuvor genannten Attribute ab, aber wir haben auch noch eine funktionale Abhängigkeit von der Personalnummer. Dies ist aber kein Schlüsselkandidat und daher ist die **3NF verletzt**.

In unserem Beispiel ist der Vorname (auch Nachname) von der Personalnummer abhängig und diese wiederum von der Projektnummer und dies nennt man dann auch transitive Abhängigkeit.

Wir müssen daher wieder zerlegen und eine neue Relation erstellen.

Projekt: $\{[Projekt_Nr: integer, Bezeichnung: string, Beginn: date, Pers-Nr: integer]\}$

Projektleiter: $\{[Pers-Nr: integer, Vorname: string, Nachname: string]\}$

4.5.4 Boyce-Codd Normalform

Die Boyce-Codd Normalform ist eine Verschärfung der 3NF.

Die Boyce-Codd-Normalform (BCNF) ist eine Weiterentwicklung der Dritten Normalform (3NF). In der Dritten Normalform kann es vorkommen, dass ein Teil eines Schlüsselkandidaten funktional abhängig ist von einem Teil eines anderen Schlüsselkandidaten. Die Boyce-Codd-Normalform verhindert diese funktionale Abhängigkeit.

Definition:

Ein Relationsschema [R] ist in Boyce-Codd Normalform (BCNF), wenn für jede [R] geltende funktionale Abhängigkeit der Form $X \rightarrow \alpha$, $X \subseteq [R]$ und $\alpha \in [R]$, wenn mindestens eine der beiden Bedingungen gilt:

- $\alpha \in X$, (dann ist die funktionale Abhängigkeit trivial)
- X ist ein Superschlüssel von [R]

Eine Relation ist in Boyce-Codd Normalform, wenn jeder Determinante ein Superschlüssel ist.

Erklärung

Eine Determinante ist eine Attributmenge (linke Seite), von der ein anderes Attribut vollständig funktional abhängig ist.

Die BCNF braucht nur dann angewendet zu werden, wenn mehrere Schlüsselkandidaten vorhanden sind und sich diese teilweise überlappen. Ist in der Relation nur ein Kandidatenschlüssel vorhanden oder es liegt keine Überlappung bei mehreren Kandidatenschlüsseln vor, befindet sich die Relation automatisch in der BCNF.

Prof. Gebhardt sagt dazu [Gebh17, S.220]: „.... die BCNF hat den Vorteil, dass sie ohne Referenz auf 2NF und 3NF auskommt und diese Konzepte damit überflüssig macht.“ Machen wir ein Beispiel indem wir die Relation aus dem Kap. 4.5.2 - nehmen und diese etwas reduzieren.

<u>Projekt-Nr</u>	<u>Pers-Nr</u>	<u>PL-Name</u>	<u>Projekt_Std</u>
4711	1	Müller	125
3050	2	Schulze	206
2020	2	Schulze	110
1234	3	Maier	154
2020	3	Maier	144
1234	4	Schwarz	154

$$\{Pers_Nr\} \rightarrow \{PL - Name\}$$

$$\{PL - Name\} \rightarrow \{Pers_Nr\}$$

$$\{Pers_Nr, Projekt_Nr\} \rightarrow \{Projekt_Std\}$$

Diese Relation ist in der 3NF, da *Pers-Nr* und *PL – Name* in einem Kandidatenschlüssels vorkommen, also Primattribut sind und somit eine der drei Bedingungen für die 3NF erfüllt ist. Da es sich um keine trivialen Funktionalen Abhängigkeiten handelt und die *Pers-Nr* keine Superschlüssel ist, wir die BCNF verletzt.

Wir normalisieren nun auf BCNF indem wir die FA auslagern, welche BCNF verletzt. Wir erhalten damit zwei neue Relationen R1 und R2.

$$R1: \{ Pers-Nr, PL-Name \}$$

$$R2 : \{ Pers-Nr, Projekt-Nr, Projekt_Std \}$$

Die *Pers-Nr* ist dabei der Fremdschlüssel.

4.5.5 Von 3NF nach Boyce-Codd und zurück

Wenn eine Relation die BCNF erfüllt, dann ist sie automatisch auch in der 3NF. Aber auch die Rückrichtung von BCNF nach 3NF gilt meistens.

Eine Relation *R* ist in BCNF, wenn diese keine überlappenden Schlüsselkandidaten hat.

Umgekehrt gilt auch, wenn die Relation *R* in 3NF aber nicht in BCNF ist, hat diese zwei sich überlappende Schlüsselkandidaten. $k_m \cap k_n \neq \emptyset$

Dies bedeutet, wenn wir eine Relation haben, welche nur einen Schlüsselkandidaten hat, so ist diese automatisch in Boyce-Codd. Auch wenn die Relation mehrere Schlüsselkandidaten hat, welche sich aber nicht überlappen, dann ist diese auch in BCNF.

4.5.6 4. und 5. Normalform

Neben den drei Normalformen und der Boyce-Codd Normalform gibt es noch die 4. und 5. Normalform, welche die Anforderungen an die Relationen weiter verschärfen, dafür aber weniger Anomalien zulassen.

Die 4NF beschäftigt sich mit *Mehrwertigen Abhängigkeiten* und die 5NF mit *Verbund-Abhängigkeiten*.

Ist die 5. Normalform erreicht, sind Einfüge-, Änderungs- und Löschanomalien ausgeschlossen. In der Praxis spielt bereits die Boyce-Codd Normalisierung keine Rolle mehr und daher kann der Normalisierungsprozess nach der 3. Normalform in der Regel als abgeschlossen betrachtet werden. Für weiterführende Studien sei auf [KeEi15, S.201-205] verwiesen.

4.5.7 Verlustfreie und abhängigkeitsbewahrende Zerlegung

In den oben beschriebenen einzelnen Normalformen ging es immer darum eine Relation so zu zerlegen, dass diese dann die jeweilige Normalform erfüllt. Nun waren die Beispiele so gewählt, dass es den Anschein hatte, dass die Zerlegung nur in der dargelegten Form erfolgen kann. Dies ist aber nicht immer der Fall und daher wollen wir uns kurz damit beschäftigen wie eine Relation zerlegt werden muss, damit wir uns nach der Zerlegung nicht weitere Probleme einhandeln. Zuerst wollen wir definieren, was eigentlich eine gültige Zerlegung ist.

Gültige Zerlegung:

Wenn wir ein Relationsschema $[R]$ in $[R_1]$ und $[R_2]$ zerlegen, dann muss gelten $[R] = [R_1] \cup [R_2]$. Wir sprechen dann von einer gültigen Zerlegung. Das heißt, nach der Zerlegung bleiben alle Attribute von $[R]$ erhalten und es kommen natürlich auch keine neuen hinzu.

Ausprägung der Zerlegung:

Die Ausprägung eines Relationsschemas $[R]$ in $[R_1]$ und $[R_2]$ ist:

$$R_1 = \Pi_{[R_1]}(R)$$

$$R_2 = \Pi_{[R_2]}(R)$$

Die bedeutet, dass R auf die beiden Relationen R_1 und R_2 projiziert wird.

Verbundtreu (verlustlose) Zerlegung:

Eine Zerlegung eines Relationsschemas $[R]$ in $[R_1]$ und $[R_2]$ ist verbundtreu oder verlustlos, wenn gilt $R_1 \bowtie R_2 = R$ für alle möglichen Ausprägungen von R .

Wenn wir die beiden zerlegten Relationsschemata mit einem Join wieder zusammenführen, ergeben sich daraus immer die ursprünglichen Daten aus R . Niemals mehr oder weniger.

Dies kann formal dadurch geprüft werden, dass gilt:

$$[R_1] \cap [R_2] \rightarrow [R_1] \in F^+$$

oder

$$[R_1] \cap [R_2] \rightarrow [R_2] \in F^+$$

D.h. das Joinattribut bestimmt eines der beiden Teilschemata. Die Existenz für einer der beiden Teilschemata eine funktionale Abhängigkeit, so ist die Verbundtreu erfüllt.

Abhängigkeitsbewahrung:

Ein weiteres Kriterium für die Zerlegung eines Relationsschemas $[R]$ ist die Abhängigkeitsbewahrung. Diese gilt jedoch nur bis 3NF und nicht mehr ab BCNF (siehe dazu Abbildung 34). Man muss nicht zuerst alle möglichen Join's durchführen, um zu zeigen, dass bei einer Zerlegung die funktio-

nalen Abhängigkeiten erhalten bleiben. Dies kann bereits innerhalb einer Relation überprüft werden. Nach der Zerlegung eines Relationsschemas [R] in verschiedene Teilschemata [R_i], kann jede FA in mindestens einer der [R_i] dargestellt werden.

Formal:

$$\left(\left(\prod_{[R1]} (F) \right) \cup \dots \cup \left(\prod_{[Rn]} (F) \right) \right)^+ = F^+$$

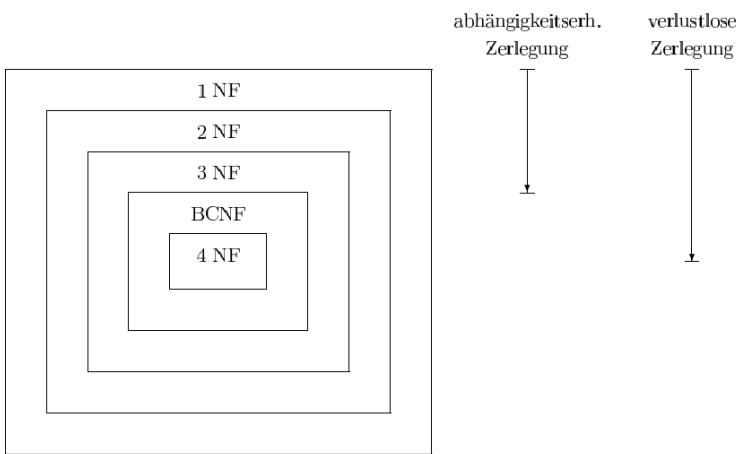


Abbildung 34: Beziehungen von Normalformen zueinander
(Quelle: [KeEi15, S.205])

5 SQL-Teil1

5.1 Einführung

SQL hat eine lange Tradition (siehe Kap. 5.1.1 Historie) und wird oft mit „Structured Query Language“ übersetzt, was aber nicht korrekt ist, da SQL eigentlich ein Eigenname ist. Dieser Titel wird aber den Möglichkeiten dieser Sprache nicht gerecht, denn sie kann viel mehr als nur Abfragen verarbeiten. Aber dazu später mehr.

SQL ist eine deklarative (deskriptive) Programmiersprache bei der der Benutzer nur angibt, welche Daten er aus den Tabellen des Datenbanksystems abfragen möchte, aber nicht wie die Abfrage durchgeführt werden soll. Man fragt in der deklarativen Programmierung (auch Lisp, Prolog...) nach dem *Was* berechnet werden soll. Im Gegensatz dazu steht bei der imperativen Programmierung (C, C++, Java...) das *Wie* im Vordergrund.

Wie die oftmals sehr komplexen Abfragen dann ausgeführt werden, dafür ist die Abfragebearbeitung (Optimierer) im DBMS zuständig (siehe Abbildung 6). Dadurch kann auch die physikalische Datenunabhängigkeit gewährleistet werden. Der Programmierer oder Benutzer hat auf Grund des deklarativen Charakters von SQL kaum Möglichkeiten den Weg der Datenbeschaffung zu beeinflussen. Er muss sich darauf verlassen können, dass seine SQL-Anweisungen vom DBMS in optimale Zugriffsprozeduren umgesetzt werden. Daran zeigt sich auch die Qualität eines Datenbanksystems.

Auch gibt es zwischen der theoretischen Betrachtung eine Relation und einem relationalen DBMS doch einige Unterschiede. Wie bereits in Kap. 3.1.2 erwähnt, können bei Relationen keine doppelten Tupel vorkommen bei den Tabellen eines relationalen DBMS kann dies aber durchaus der Fall sein. Wir sprechen hier auch nicht mehr von Tupel, oder Attributen in Relationen, sondern wie in Tabellen üblich von Zeilen und Spalten der Tabelle.

SQL ist wie seine Grundlage – die Relationale Algebra – eine mengenorientierte Datenbanksprache. Mit einer SQL-Anweisung werden in der Regel mehrere Tabellenzeilen angesprochen. Der wesentliche Vorteil einer mengenorientierten Verarbeitung ist, dass komplexe Aufgabenstellungen mit einer Anweisung erledigt werden können. Andererseits birgt es auch die Gefahr, dass es bei komplexen Abfragen oft schwierig ist, das Ergebnis gedanklich nachzuvollziehen.

5.1.1 Historie

Der Ursprung von SQL begann in den Forschungslaboren der Firma IBM in denen im Jahr 1974 der DBMS-Prototyp „System R“ entwickelt wurde. Für dieses DBMS wurde die Sprache „SEQUEL“ (Structured English Query Language) entworfen, welche nach einigen Änderungen letztendlich dann SQL genannt wurde.

Aber nicht IBM, sondern die Firma ORACEL war es, die 1979 das erste kommerzielle DBMS auf den Markt brachte. IBM zog dann zwei Jahre später mit SQL/DS nach weiteren zwei Jahren mit DB2 nach.

5.1.2 SQL-Standards

Sehr schnell war nach dem Erfolg von SQL klar, dass eine Standardisierung erfolgen musste. Im Jahre 1986 wurde vom amerikanischen Normengremium (ANSI) eine erste SQL-Norm verabschiedet. Dieser ANSI-Norm hat sich später die ISO (International Standards Organization) angegeschlossen. Nachfolgend sind die wesentlichen Standardisierungen der letzten Jahre kurz aufgeführt.

SQL-Standard	Informationen und wichtige Neuerungen
SQL-86	Erste Normierung durch ANSI
SQL-89	Normierung von Basisoperationen, Definition von referentiellen Integrität
SQL-92 (SQL 2)	Strukturierung Systemkatalog, Änderungsmöglichkeiten bei der Definition von Datenbankobjekten, dynamisches SQL
SQL-99 (SQL 3)	Rekursive Anfragen, Trigger, Objektrelationale Systeme und Objektorientierte Systeme, nicht mehr nur auf Relationale Algebra beschränkt (Operationen ohne Entsprechung in RA)
SQL-2003	XML-Funktionen; Windows Funktionalität; SQL-Merge
SQL-2006	Aufruf von SQL aus XML (XQuery)
SQL-2008	"INSTEAD OF"-Trigger und "TRUNCATE"-Statement
SQL-2011	System-Versionierung (alte Zeilen bleiben auch nach Update und Insert erhalten)
SQL-2016	Zeilenmustererkennung identifiziert, Gruppen von Zeilen, deren Abfolge einem Muster entspricht; JSON-Unterstützung; Trigonometrische und logarithmische Funktionen

Tabelle 1: SQL-Standard

Trotz aller Normierungsbestrebungen spricht fast jedes Datenbankprodukt sein eigenes SQL oder lehnt sich an einen der oben beschriebene Standards an. SQL ist leider nicht gleich SQL.

5.1.3 PostgreSQL - Das System zum Üben

Neben MySQL ist PostgreSQL eines der populärsten Datenbanksysteme im Bereich der freien Software (Open Source).

PostgreSQL unterstützt weitgehend die ANSI-Standards SQL92, SQL99 und SQL2003. PostgreSQL unterstützt seit langer Zeit Programmierkonstrukte wie gespeicherte Prozeduren, Trigger und Cursor und verfügt über Schnittstellen zu allen modernen Programmiersprachen und Script-sprachen (wie Java, C/C++, PHP, Perl und Python).

Die Leistungsfähigkeit und Flexibilität von PostgreSQL ist in vielen Bereichen mit denen von kommerziellen Datenbanken (wie IBM DB2 oder Oracle) verglichen werden. Zu den Benutzern gehören weltweit bekannte Firmen wie Cisco, Sony, Apple, BASF, NASA, die UNO, zahlreiche Regierungsorganisationen und Hochschulen. Wegen der liberalen OpenSource-Lizenz kann der Quellcode sowohl innerhalb von Open Source als auch innerhalb kommerzieller Systeme modifiziert werden.

PostgreSQL erlaubt den Benutzern auch, das System um selbstdefinierte Datentypen, Operatoren und Funktionen zu erweitern.

Die Geschichte von PostgreSQL

PostgreSQL ist das wahrscheinlich am weitesten entwickelte Open-Source-System im Internet. Es ist aus einer Datenbankentwicklung (Ingres-Projekt) an der University of California in Berkeley entstanden. Im Jahr 1986 begann die Weiterentwicklung von Ingres unter dem Namen Post-Ingres. Dieser Name veränderte sich während der Überarbeitung in Postgres und bekam auch eine andere Codebasis als die des ursprünglichen Ausgangssystems. Im Jahr 1989 wurde die erste Version von Postgres fertig gestellt und der Öffentlichkeit präsentiert. Zuerst hatte Postgres ein eigene Sprache POSTQUEL, welche dann im Jahr 1994 um einen SQL-Interpreter ergänzt und unter dem Namen

Postgres95 als Open-Source-System veröffentlicht wurde. Die Umbenennung in PostgreSQL kam 1996, als auch die Weiterentwicklung durch die zusammenarbeitenden Programmierer aus der ganzen Welt begann. Im Jahr 2005 wurde die Version 8 veröffentlicht, die erstmals das Betriebssystem Windows von Microsoft unterstützt und zahlreiche Verbesserungen der Systemleistung enthält. Die ersten professionellen Neuerungen (Benchmarks) vom August 2007 zeigten, dass PostgreSQL gegenüber Oracle nur ca. 12% langsamer ist, dies jedoch bei deutlich geringeren Entwicklungs- und Wartungskosten.

PostgreSQL als Client/Server System

Offene Datenbanktechnologien ermöglichen den Austausch des Server-DBMS gegenüber der unabhängigen Client-Applikation. PostgreSQL arbeitet nach dem Client/Server-Konzept. Es realisiert ein funktional verteiltes System, in dem zwei unabhängige Prozesse (z. B. zwei Programme) über eine definierte Schnittstelle miteinander kommunizieren.

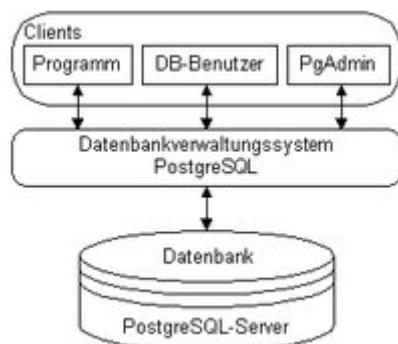


Abbildung 35: Client/Server System
(vergl. [2])

Das Datenbankverwaltungssystem ist die Schnittstelle zwischen Clients und Server. Der Server-Prozess (PostgreSQL-Server) läuft permanent am Datenbankserver, auf dem die eigentlichen Datenbankdateien gespeichert sind. Der Datenbankserver stellt Dienstleistungen zur Verfügung und die Clients nutzen die Serverdienste. Eine Datenbankanwendung (Client) nimmt z. B. durch eine Datenbankanfrage Dienste des PostgreSQL-Servers in Anspruch. Der Datenbankserver führt die Abfrage über die Datenbank aus und sendet eine selektierte Datenmenge an den Client zurück.

Der grafische Administrator pgAdmin von PostgreSQL fungiert auch als PostgreSQL Client. Auch Webapplikationen oder beliebige Programme, welche die ODBC (Open Database Connectivity) oder JDBC (Java Database Connectivity) Schnittstellen unterstützen (z. B. Office Software), können als Clients agieren. Server und Clients können auf gleichen oder (in der Regel) auf unterschiedlichen Rechnern unter unterschiedlichen Betriebssystemen laufen. Im zweiten Fall erfolgt dann die Kommunikation über Netzwerkverbindungen (TCP-IP). Der PostgreSQL-Serverprozess (postmaster) hat am TCP-IP die Portadresse 5432 (siehe [2]).

5.1.4 Bestandteile von SQL

Wie schon erwähnt, ist SQL viel mehr als eine Abfragesprache. Es setzt sich aus folgenden Sprachteilen zusammen:

- **Data Definition Language (DDL):** Diese dient zum Anlegen, Ändern und Löschen von Datenbanken, Tabellen usw. Hier wird das Schema definiert.
- **Data Manipulation Language (DML):** Diese dient zum Anlegen, Ändern und Löschen von Daten in einer Tabelle.
- **Data Control Language (DCL):** Diese dient für Administratoren zur Verwaltung von Berechtigungen (z.B. Lese- und Schreibberechtigungen).
- **Data Query Language (DQL):** Diese dient zum Abfragen von Daten aus den Tabellen. Hiermit wird SQL als Abfragesprache verbunden.
- **Transaction Control Language (TCL):** Enthält Sprachelemente zum Arbeiten mit Transaktionen. Wird oft auch der DML oder der DCL zugeordnet.

5.2 Datenbanken und Tabellen definieren (DDL)

5.2.1 CREATE Database

Nach der doch etwas längeren Einführung in SQL wollen wir nun damit beginnen unserer Datenbank und alle im RM definierten Tabellen, sowie die Verbindungen zwischen den Tabellen anzulegen. SQL kennt in der DDL dazu verschiedene Befehle wie CREATE, ALTER, DROP... usw. Alle nachfolgenden SQL-Befehle sind an PostgreSQL angelehnt.

Beginnen wir mit dem Anlegen unseres Softwarehauses und dazu müssen wir zuerst die Datenbank definieren. Dazu dient der Befehl:

```
CREATE DATABASE DatenbankName
```

Um nun konkret die Datenbank für unser Softwarehaus zu erstellen gehen wir wie nachfolgend vor:

```
CREATE DATABASE Softwarehaus
  WITH
    OWNER = postgres
    ENCODING = 'UTF8'
    LC_COLLATE = 'German_Germany.1252'
    LC_CTYPE = 'German_Germany.1252'
    TABLESPACE = pg_default
    CONNECTION LIMIT = -1;
```

Im Prinzip reicht die erste Zeile des CREATE-Befehls aus um die Datenbank anzulegen. Der Rest zeigt auf, welche Parameter Postgres Default mäßig noch anlegt. Von diesen Parametern interessiert uns derzeit nur der OWNER, welcher besagt, dass die DB unter dem Standard-Besitzer „postgres“ angelegt wurde.

5.2.2 CREATE Schema

PostgreSQL kennt den Begriff des Schemas. Ein SQL-Schema ist ein dynamischer Sichtbarkeitsbereich für geschachtelte (lokale) SQL-Objekte (Tabellen, Views, Indizes, Funktionen, Regeln ...), welche unterschiedliche Namen erhalten. Dies ist aber nicht gleichzusetzen mit dem Begriff des Schemas für den Aufbau einer Relation (Kap. 3.1.2).

Schemata dienen zur Strukturierung innerhalb einer Datenbank. Unter einem Schema können mehrere Tabellen angelegt werden und in einer Datenbank kann es mehrere Schemata geben. Schemata sind ähnlich der Directory-Struktur von Betriebssystemen, nur dass Schemata nicht verschachtelt werden können. Tabelle können in verschiedenen Schemata dieselben Namen erhalten. Es kann z.B. im Schema <public> (wird standardmäßig von Postgres angelegt) und in <myschema> eine Tabelle mit dem Namen *Mitarbeiter* geben. Auch können Tabellen über mehrere Schemata hinweg verbunden (join) abgefragt werden. Dazu muss dann der Name des Schemas und der Name der Tabelle angegeben werden (*public.Mitarbeiter*).

Der Befehl dazu lautet:

```
CREATE SCHEMA SchemaName
  AUTHORIZATION postgres;
```

Es gibt verschiedene Gründe Schemata zu verwenden:

- Verschiedene Benutzer können dieselbe Datenbank verwenden, ohne sich gegenseitig zu behindern.
- Datenbank-Objekte in verschiedene logische Gruppen aufzuteilen, um diese leichter zu handhaben.
- Die Objekte von Fremdanwendungen können in ein getrenntes Schema verlagert werden, sodass diese nicht mit Namen anderer Objekte kollidieren (weil diese zufällig gleich heißen).

5.2.3 Tabellen anlegen und verwalten

Nun kommen wir zu einem der wichtigsten Teile innerhalb der DDL. Das Anlegen und Verwalten von Tabellen und damit zu der Umsetzung unseres ER-Modells und der daraus definierten Relationen in eine Datenbank.

Hierbei müssen wir natürlich auch die verschiedenen Bedingungen (engl. Constraints) berücksichtigen, welche wir in der Modellierungsphase definiert haben. Daher wollen wir der Verwaltung von Tabellen an dieser Stelle einen großen Raum geben.

Nun sollte man nicht damit beginnen wild irgendwelche Tabellen zu erzeugen, sondern - wie bereits mehrfach erwähnt - sollte der Entwurfsprozess einer Datenbank strukturiert durchgeführt werden (siehe **Abbildung 7**). Es kann ansonsten später zu Problemen (z.B. Performanceprobleme) und erhöhten Kosten führen.

5.2.3.1 CREATE Table

Das Anlegen einer Tabelle erfolgt mit dem Befehl:

```
CREATE TABLE TabellenName
(
    Attribut_1 Domäne_1,
    ...
    Attribut_n Domäne_n );
```

Dem Namen der Tabelle folgen die Attribute mit den entsprechenden den Domänen (Datentypen). Für diese Domänen gibt es in PostgreSQL verschiedene vordefinierte Datentypen.

Für relationale Datenbank gibt es im Prinzip drei fundamentale Datentypen welche als Domänen verwendet werden. Dies sind Zahlen, Zeichenketten und der Datumstyp.

Nachfolgend sehen Sie eine Liste mit einigen wichtigen Datentypen (weitere finden Sie im Anhang):

Datentyp	Wertebereich / Beschreibung
integer, int	Ganzzahlwert zwischen -2147483648 to +2147483647 (es gibt auch <i>smallint</i> , <i>bigint</i>)
decimal(p,s) numeric(p,s)	Zahl mit vorgegebener Vor- und Nachkommastellen (p Ziffern und davon s nach dem Komma)
serial	Ganzzahlwert mit Autoinkrement (integer). Es gibt auch <i>smallserial</i> und <i>bigserial</i>
character(n) char(n)	Zeichenkette mit fixer Länge n, wird mit Leerzeichen aufgefüllt
character varying(n) varchar(n)	Zeichenkette mit variabler maximaler Länge n
date	Hier wird eine Zeichenkette spezielle für das Datum aufbereitet. Das Format ist 2019-03-28 (interne Speicherung)

Tabelle 2: Auswahl an Datentypen

Die Wahl des „richtigen“ Datentyps für jedes Attribut einer Tabelle hat verschiedene Vorteile und es sollte sich daher im Vorfeld (bei der Modellierung) Gedanken dazu gemacht werden. Kurz sollen folgende Vorteile erwähnt werden:

- Operationen auf Spalten mit demselben Datentyp liefern auch konsistente Ergebnisse.
- Wird z.B. ein einfacher Datentyp gewählt, kann eine verdichtete Abspeicherung erfolgen.
- Bei der Wahl des geeigneten Datentyps erfolgt eine effiziente Speicherung, was bei Auffangen einen Performancevorteil bedeuten kann.

Es gibt Datentyp wie beispielsweise VARCHAR (n) welche zum SQL-Standard gehören, der Type *Text* hingegen ist ein PostgreSQL spezifischer Datentyp. Die Verwendung der verschiedenen Datentypen wie z.B. SERIAL erfolgt in den folgenden Abschnitt.

Darüber hinaus unterstützt PostgreSQL die gängigen SQL Datentypen zur Speicherung von Datums- und Uhrzeitwerten, Gleit- und Fließkommazahlen, sowie großen Objekten (LOBs). Auch anwendungsspezifische Datentypen für bspw. geometrische Objekte oder Netzwerkadressen sind in PostgreSQL vordefiniert. Zusätzlich ist es in PostgreSQL auch möglich eigene Datentypen (CREATE TYPE) zu definieren.

Nachfolgend ein Beispiel für die Definition der Tabelle Mitarbeiter:

```
CREATE TABLE Mitarbeiter
(
  PERS_Nr integer NOT NULL,
  Vorname character varying,
  Nachname character varying,
```

```
Geb_Name varchar,  

Geb_Datum date,  

Geschlecht char(1),  

Eintrittsdatum date  

);
```

In diesem Beispiel einer Tabellendefinition sehen wir bereits eine Besonderheit, nämlich eine weitere Angabe zur Personalnummer die Definition NOT Null. Hierbei handelt es sich um eine Bedingung, welche wir beim Anlegen der Personalnummer mitgeben.

PostgreSQL kennt eine Reihe von Bedingungen, welche nachfolgend aufgeführt sind.

Constraint	Verwendung
NOT NULL	Stellt sicher, dass dieses Attribut keinen NULL-Wert enthält.
UNIQUE	Jeder Wert in dieser Spalte der Tabelle ist eindeutig.
PRIMARY Key	Dieses Attribut stellt den Primärschlüssel dar und identifiziert dadurch einen Datensatz eindeutig.
FOREIGN Key	Fremdschlüssel zeigt auf ein Attribut (Spalte) in einer anderen Tabelle (stellt zu dieser eine Verbindung her).
CHECK Bedingung	Mit dieser Angabe kann sichergestellt werden, dass ein Wert eines Attributs eine bestimmte Bedingung erfüllt.
DEFAULT wert	Mit dieser Bedingung kann ein Attribut (Spalte) automatisch mit einem Wert vorbelegt werden, wenn von außen kein Wert zugewiesen wurde.

Tabelle 3: Constraint

Allgemein ist das Anlegen einer Tabelle mit Constraints folgendermaßen aufgebaut.

```
CREATE TABLE TabellenName  

(  

Attribut_1 Domäne_1,  

...  

Attribut_n Domäne_n,  

Constraint_1,  

...  

Constraint_m  

);
```

Nehmen wir nun wieder unsere Mitarbeiter-Tabelle und überlegen uns welche Bedingungen hier sinnvoll wären.

```
CREATE TABLE Mitarbeiter  

(  

Pers_Nr integer,  

Vorname varchar,  

Nachname varchar NOT NULL,  

Geb_Name varchar,  

Geb_Datum date,  

Geschlecht char(1) NOT NULL,
```

```

    Eintrittsdatum date NOT NULL,
    PRIMARY KEY (Pers_Nr),
    CHECK (Geschlecht IN ('m', 'w'))
);

```

Mit obiger Definition der Tabelle gelten nun verschiedene Bedingungen. Die Attribute *Nachname*, *Eintrittsdatum* und *Geschlecht* dürfen nicht NULL sein. Beim *Geschlecht* ist nur „m“ oder „w“ zugängig und die *Personalnummer* wurde als Primärschlüssel definiert.

Erweitern wir nun die Attribute dieser Tabelle z.B. noch um die Attribute *Gehalt* und *Skill*, so können wir beispielsweise festlegen, dass das Gehalt sich nur in bestimmten Grenzen bewegen darf und dass beim *Skill* eine Vorbelegung stattfindet.

```

CREATE TABLE Mitarbeiter
(
  ...
  Skill varchar(5) default 'PR',
  Gehalt int,
  PRIMARY KEY (Pers_Nr),
  CHECK (Geschlecht IN ('m', 'w')),
  CHECK (Gehalt > 20000 and Gehalt < 150000)
);

```

Nun wollten wir uns noch der Verbindung zweier Tabellen zuwenden. Hier kann bereits beim Anlegen dieser Tabellen angegeben werden, wie diese zueinander in Verbindung stehen. Dies erfolgt mit der Definition des FOREIGN KEY. Hierzu erstellen wir die Tabelle *Abteilung* und verbinden diese mit der Tabelle *Mitarbeiter*.

```

CREATE TABLE Abteilung
(
  Abt_Bez_kurz varchar (8),
  Abt_Bez_lang varchar NOT NULL,
  Standort varchar NOT NULL,
  PRIMARY KEY (Abt_Bez_kurz)
);

```

Die Verbindung zwischen der Tabelle *Mitarbeiter* und *Abteilung* ist die Personalnummer des Mitarbeiters und diese identifiziert eindeutig die Abteilung in der dieser arbeitet. Wie wir in Kap. 3.2.3 aufgezeigt haben, kann nun der Primärschlüssel der Tabelle *Abteilung* als Fremdschlüssel bei *Mitarbeiter* eingetragen werden.

Hinweis:

Das Erstellen von Tabellen mit Beziehungen kann nicht einfach willkürlich erfolgen, da die Tabellen auf welche die Verbindungen referenzieren bereits vorhanden sein müssen, bevor die Referenz erfolgen kann. Es müssen daher zuerst alle Tabellen angelegt werden, von denen keine Beziehungen ausgehen und dann in einem weiteren Schritt werden die Tabellen angelegt welche auf die ersten verweisen.

Der einfachere Weg ist es, zuerst alle Tabellen ohne Fremdschlüsselverweise anzulegen, um dann

in einem zweiten Schritt alle Beziehungen einzufügen. Dann muss keine bestimmte Reihenfolge eingehalten werden.

Wir verändern dazu die Tabelle *Mitarbeiter* durch den Befehl ALTER und fügen die Abteilungskurzbezeichnung (Abt_Bez_kurz) aus der Tabelle *Abteilung* als Fremdschlüssel ein. Nun besteht die Verbindung zwischen diesen beiden Tabellen, welche auch bei einer Abfrage verwendet werden kann.

Grundsätzlich würde es ausreichen, wenn wir nur den Primärschlüssel der Tabelle *Abteilung* in die Tabelle *Mitarbeiter* eintragen würden. Der SQL-Befehl FOREIGN Key ist zur Kennzeichnung nur für den Optimizer erforderlich, damit dieser eine optimale Zugriffsstrategie errechnen kann.

5.2.3.2 ALTER Table

```
ALTER TABLE Mitarbeiter
ADD arbeitet_in varchar (8),
ADD FOREIGN KEY(arbeitet_in) REFERENCES Abteilung;
```

Der Befehl ALTER wird dann verwendet, wenn wir an einer bestehenden Tabelle die Struktur verändern möchten. Er kann daher auf verschiedene Weise eingesetzt werden. Eine Möglichkeit haben wir gerade kennen gelernt, indem wir ein Attribut (Spalte) zur Tabelle *Mitarbeiter* hinzugefügt und zusätzlich dieses Attribut als Fremdschlüssel gekennzeichnet haben. Es können aber auch Tabellen oder Attribute umbenannt oder Constraints hinzugefügt werden. Wir werden im Laufe dieses Kapitels noch einige Variationen kennlernen, aber ansonsten sei auf die Dokumentation von PostgreSQL [2] verwiesen. Nachfolgend noch ein paar weitere Beispiele zu ALTER.

```
ALTER TABLE TabellenName RENAME TO TabellenName_neu;
ALTER TABLE TabellenName ADD UNIQUE(Attribut);
ALTER TABLE TabellenName ADD CHECK(Abfrage);
...
```

5.2.3.3 DROP

Natürlich kann man nicht nur Datenbanken, Tabellen und Attribute anlegen, sondern auch löschen. Auch hier wirkt dieser SQL-Befehl auf verschiedenen Ebenen der Datenbank. Auch dazu nachfolgend einige Beispiele.

```
DROP TABLE Mitarbeiter;
ALTER TABLE Mitarbeiter DROP Skill;
```

Mit dem ersten Befehl wird die Tabelle *Mitarbeiter* gelöscht und mit dem zweiten das Attribut *Skill* aus der Tabelle *Mitarbeiter*. Nebenbei erwähnt, kann auch mit `DROP DATABASE` eine komplette Datenbank gelöscht werden. `DROP` kennt noch mehrere Varianten, aber uns reichen derzeit die aufgeführten.

Nun haben wir das Werkzeug zusammengetragen um Datenbanken und Tabellen anzulegen und zu verwalten. Nun müssen wir aber auch Daten in die Tabelle einfügen und diese verwalten und dazu dient uns die Data Manipulation Language (DML) als Teil von SQL.

5.3 Daten in Tabellen verändern (DML)

Um nun Tabelleninhalt zu verändern, existieren drei verschiedene Möglichkeiten:

1. Einfügen einer neuen Zeile
2. Modifizieren einer bereits bestehenden Zeile
3. Löschen einer Zeile

5.3.1 Eine neue Zeile einfügen

Mit dem SQL-Befehl `INSERT` können Daten in eine Tabelle eingefügt werden. Dazu ist natürlich zu beachten, dass Daten gemäß der Definition des Datentyps eines Attributs eingefügt werden. Der `INSERT` Befehl hat folgende allgemeine Form:

```
INSERT INTO TabellenName
VALUES (Wert_1, Wert_2, Wert_3);
```

Hierbei werden nun die angegebenen Werte in die Tabelle *Tabellenname* eingefügt. Bei dieser Variante des `INSERT`-Befehls muss die Reihenfolge der Werte, der Reihenfolge der Spaltenwerte der Tabelle entsprechen und kein Wert darf ausgelassen werden.

Als Beispiel wollen wir in unsere Tabelle *Mitarbeiter* ein Tupel (Zeile) einfügen. Die Zeichenketten geben wir in einfachen Anführungsstrichen an.

```
INSERT INTO Mitarbeiter
VALUES (1, 'Hans', 'Müller', Null, '1975-03-12', 'm', '2014-07-01', default, 48000);
```

Wie schon des Öfteren erwähnt, wird ein Wert, der uns derzeit unbekannt ist, mit `NULL` angegeben und an der entsprechenden Stelle im Tupel eingefügt.

Bei einer anderen Variante des Insert-Befehls gibt man explizit die Werte an, welche eingefügt werden sollen. Hier ist natürlich zu beachten, dass die gesetzten Constraints nicht verletzt werden (z.B. fehlender Wert bei `NOT NULL`). Die Reihenfolge der Werte ist frei wählbar.

```
INSERT INTO Mitarbeiter (pers_Nr, nachname, geschlecht, eintritts-datum)
values (3, 'Klein', 'm', '20.03.2012')
```

Als Hinweis sei an dieser Stelle nochmals vermerkt, dass die neu eingefügte Zeile in einer Tabelle keine vorhersagbare Position einnimmt. Dies ist absolut zufällig und die Zeilen einer Tabelle besitzen keine natürliche Reihenfolge.

Mit `SELECT`-Befehl (welchen wir in Kap. 7 noch ausführlich behandeln werden) können wir die Daten wieder aus der Tabelle auslesen und überprüfen. Die einfachste Form des `SELECT`-Befehls lautet:

```
SELECT * FROM TabellenName
```

```
SELECT * FROM Mitarbeiter;
```

5.3.2 Bestehende Daten verändern

Die Modifikation von bereits bestehenden Zeilen wird dann notwendig, wenn sich bestimmte Werte ändern (z.B. Nachname eines Mitarbeiters), oder wenn falsche Daten eingefügt wurden, welche nun nachträglich angepasst werden müssen. Der SQL-Befehl dazu lautet UPDATE:

```
UPDATE TabellenName
SET Attribut = Wert, Attribut1 = Wert1, ...
WHERE <Prädikat>
```

```
UPDATE TabellenName
SET Nachname = 'Maier'
WHERE Pers_Nr = 1;
```

Lässt man die WHERE-Klausel weg, so wird das Update für alle Zeilen durchgeführt. Auch ist es möglich dem Attributwert nicht nur einen konstanten Wert zuzuweisen, sondern auch das Arbeiten mit arithmetischen Ausdrücken (*Gehalt = Gehalt * 1,08*) ist erlaubt.

Weiterhin ist noch zu beachten, dass, wenn man z.B. alle Attribute sucht, bei denen ein bestimmter Wert bisher noch nicht belegt wurde (NULL), so lautet der Befehl:

```
...
WHERE Gehalt is NULL
```

5.3.3 Löschen von Daten

Die letzte Änderungsoperation betrifft das Löschen von Zeilen in einer Tabelle.

Hier verwenden wir den SQL-Befehl DELETE. Die allgemeine Form dazu lautet:

```
DELETE FROM TabellenName
WHERE <Prädikat>
```

Wollen wir nun beispielsweise einen Mitarbeiter löschen, welcher unser Softwarehaus verlassen hat, so schreiben wir:

```
DELETE FROM Mitarbeiter
WHERE Pers_Nr = 5;
```

Bei diesem Befehl hat das Weglassen der WHERE-Klausel fatale Folgen, denn dann würden alle Zeilen der Tabelle gelöscht werden.

5.3.4 Daten in verbundenen Tabellen löschen

Eine Besonderheit haben wir beim Löschen von Daten Kap.5.3.3 bewusst ausgelassen, nämlich was passiert wenn Daten gelöscht werden sollen, zu denen es Abhängigkeiten gibt (Fremdschlüsselverbindung zu weiterer Tabelle vorhanden). Man unterscheidet dabei drei Möglichkeiten und dazu schauen wir uns wieder verschiedene Beispiele aus unserem Softwarehaus an.

Diese nachfolgend beschriebenen Löschregeln werden vom DBMS dann angewandt, wenn eine Zeile aus der Tabelle gelöscht werden soll, deren Primärschlüssel in der verbundenen Tabelle als Fremdschlüsselwert enthalten ist. Das Ziel dieser Überwachung ist es, die referentielle Integrität sicher zu stellen. Wir sagen dem DBMS mit diesen Löschregeln, wie es auf die Verletzung der referentiellen Integrität reagieren soll.

5.3.4.1 Abhängige Daten auf NULL setzen



In unserem ersten Beispiel betrachten wir eine klassische 1:N-Beziehung ohne hierarchische Abhängigkeit. Soll nun, aus für uns nicht näher nachvollziehbaren Gründen, eine Abteilung gelöscht werden zu der noch Mitarbeiter zugeordnet sind, so dürfen diese natürlich nicht mit gelöscht werden. Da wir im Moment nicht wissen in welcher Abteilung diese Mitarbeiter zukünftig arbeiten werden, werden die Fremdschlüssel in der Spalte „*arbeitet_in*“ auf NULL gesetzt.

Nun müssen wir, wie bereits erwähnt, dem DBMS noch mitteilen, dass wir für diese Beziehung die Löschregel „NULL setzen“ wünschen und dies erfolgt bereits beim CREATE-Befehl in der Angabe des Constraint.

```

CREATE TABLE Mitarbeiter
(
  ...
  FOREIGN KEY(arbeitet_in) REFERENCES Abteilung on Delete Set
  NULL ;
  
```

5.3.4.2 Die abhängigen Daten werden mit gelöscht



Betrachten wir nun die Tabellen *Kunde* zu *Auftrag*. In diesem Beispiel besteht eine hierarchische Abhängigkeit zwischen Kunde und Auftrag. Ein Auftrag existiert im Normalfall nicht ohne einen Kunden, daher sollten, wenn ein Kunde gelöscht wird, auch alle zugehörigen Aufträge gelöscht werden. Dies erreichen wir durch die Löschregel CASCADE (kaskadierendes Löschen).

```

CREATE TABLE Auftrag
(
  ...
  FOREIGN KEY(erteilt) REFERENCES Kunde on Delete CASCADE;
  
```

5.3.4.3 Löschen der abhängigen Daten abweisen

Wurde bei der Definition des Fremdschlüssels keine Löschregel angegeben, so wird per Default vom DBMS RESTRICT gesetzt. Dies bedeutet, dass, wenn eine Fremdschlüsselbeziehung existiert, das Löschen eines Datensatzes aus der referenzierten Tabelle abgewiesen wird. Dies könnte in unserem Softwarehaus an der Stelle Projekt und Leistung zum Einsatz kommen. Denn welche Leistungen einem Projekt aus einem Auftrag zugeordnet sind, dürfen im Regelfall nicht verlorengehen. Sollte versucht werden ein Projekt zu löschen, dann müsste dies in diesem Fall verhindert werden.

```

CREATE TABLE Leistung
(
  ...
  ADD FOREIGN KEY(zu_Projekt) REFERENCES Projekt on Delete RESTRICT;
  
```

5.3.4.4 Hinweise zu Löschregeln

Das Löschen von abhängigen Datensätzen in komplexen Datenmodellen ist eine sehr schwierige Angelegenheit und mit äußerster Vorsicht anzuwenden. Hier sind in der Regel nicht nur die Beziehungen zweier Tabellen betroffen, sondern es existieren üblicherweise Beziehungsstrukturen (siehe *Kunde, Auftrag, Leistung, arbeitet_an, Projekt*). Hier muss sorgfältig nach Analyse der Miniwelt abgewogen werden, welche Löschregeln anzuwenden sind. Oftmals wird daher das kaskadierende Löschen nicht erlaubt und stattdessen eine zusätzliche Spalte in die referenzierende Tabelle eingefügt, an der erkannt werden kann, ob der abhängige Datensatz noch aktiv ist. Dies ist, wie wir zwischenzeitlich natürlich wissen, eine höchst fragwürdige Aktion, da nun die Datenintegrität per Programm überwacht werden muss.

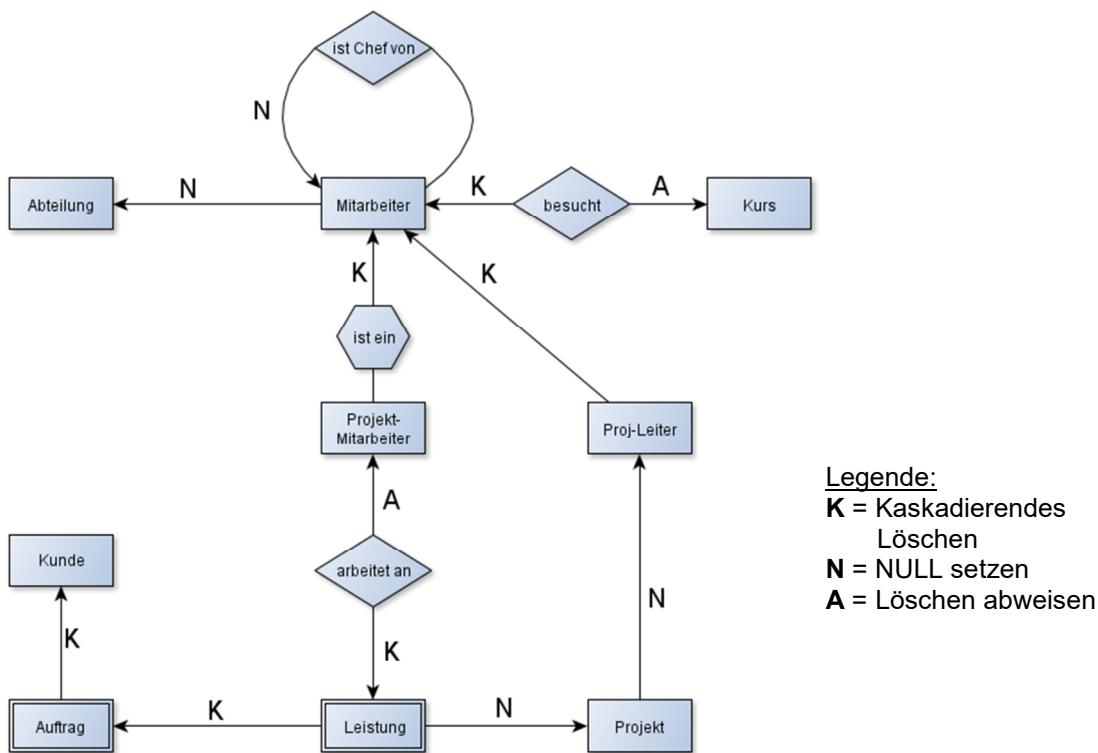


Abbildung 36: Löschregeln Softwarehaus

5.3.4.5 Regeln für den Update

Auch das Verändern von Daten in Tabellen mit Fremdschlüsselbeziehungen bedarf bestimmter Regeln, die als `UPDATE ON...` beim `CREATE`-Befehl mitangegeben werden können. Wird der Primärschlüssel geändert, so sollen im Regelfall auch alle Fremdschlüsseleinträge der referenzierten Tabelle angepasst werden. Dies wird mit `UPDATE ON CASCADE` erreicht. Wird nichts angegeben, so zieht wieder die `RESTRICT`-Bedingung und ein Update wird abgelehnt. Da ein Primärschlüssel nicht `NULL` sein darf, gibt `SET NULL` als Regel wenig Sinn.

6 Die Relationale Algebra (eine formale Sprache)

6.1 Definition

Bei der Relationalen Algebra handelt es sich um eine formale Sprache zur Formulierung von Operationen (Abfragen) auf Relationen. Wie bereits erwähnt hat Codd als Mathematiker dieses sehr formale Konstrukt mit acht Operatoren definiert. Wir wollen nachfolgend alle Operationen kurz besprechen, denn diese dienen uns als Grundlage für das Verständnis der Abfragesprache SQL. Wenn wir diese Operationen auf Relationen ausführen erhalten wir als Ergebnis wieder eine Relation. Dies bedeutet, die Operationen sind in sich abgeschlossen.

Formale Definition:

Es seien R1 und R2 Relationen. Dann sind

$R' := \langle op \rangle_{\langle Parameter \rangle} R_1$, sowie

$R'' := R_1 \langle op \rangle_{\langle Parameter \rangle} R_2$

ebenfalls Relationen

Dies bedeutet, dass ein Operator ($\langle op \rangle$) die Relation R1 als Eingabe nimmt und heraus kommt eine neue Relation R'. Es können auch zwei Relationen mit einem Operator verknüpft werden und als neue Relation erhalten wir dann R''. Jedem Operator können noch Parameter mitgegeben werden.

Nehmen wir an „ α “ wäre die Operation auf eine Relation R, dann erhalten wir als Ergebnis eine neue Relation z.B. „ $R_1 \alpha$ “.

$\alpha(\text{Mitarbeiter_Softwarehaus}) = R_1$

Auf diese neu erhaltene Relation „ $R_1 \alpha$ “ können wir nun wieder eine Operation durchführen und diese miteinander verknüpfen. So können wir auch sehr komplexe Abfragen bilden.

Zur Erklärung verwenden wir unsere Relation „Mitarbeiter_Softwarehaus“

Pers-Nr	Vorname	Nachname	Geschlecht	Geb-Name	Eintritts-Datum	Skill	Gehaltstufe
1	Hans	Müller	m	NULL	1.7.2001	PR	It2
2	Rita	Schulze	w	NULL	1.11.2007	DBA	It3
3	Werner	Maier	m	NULL	1.1.2010	Test	It2
4	Karin	Schwarz	w	Klein	1.3.2005	PR	It2

6.2 Selektion

Mit der Selektion wählt man aus der Ausgangsrelation die Tupel aus, welche die angegebene Bedingung erfüllen. Die Bedingung wird als Prädikat zu der jeweiligen Operation geschrieben. Die Selektion wird mit dem griechischen Buchstaben Sigma „ σ “ (kennen wir aus der Mathematik als Standardabweichung) abgekürzt. Nun können wir mit dieser Operation z.B. auf unsere Relation „Mitarbeiter_Softwarehaus (M_S)“ alle Tupel bestimmen, bei denen das Geschlecht weiblich ist. Allgemein:

$R' := \sigma_P(R) \subseteq R$
 $P: [R] \rightarrow \text{bool}$

Das Prädikat ist eine Funktion und ist definiert auf dem Schema von „R“ ($[R]$) und wird durch eine Formel F mit folgenden Bestandteilen beschrieben:

- Attributnamen der Argumentrelation R oder Konstanten als Operanden
- arithmetische Vergleichsoperatoren $<= > \leq \neq \geq$
- logische Operatoren: $\wedge \vee \neg$ (und, oder, nicht)

Das Ergebnis ist entweder „wahr“ oder „falsch“ und ist somit ein boolescher Ausdruck.

Beispiel:

$\sigma_{\text{Geschlecht}='w'}(M_S)$

Wir erhalten damit folgende Tupel:

Pers-Nr	Vorname	Nachname	Geschlecht	Geb-Name	Eintritts-Datum	Skill	Gehaltstufe
2	Rita	Schulze	w	NULL	1.11.2007	DBA	It3
4	Karin	Schwarz	w	Klein	1.3.2005	PR	It2

Die neue Relation hat nun zwei Tupel als Ergebnismenge und man spricht dann auch von der Mächtigkeit 2. Bei der Selektion erhalten wir wieder alle Attribute der Relation.

6.3 Projektion

Durch die Projektion erhalten wir nun die Möglichkeit einzelne Attribute (Spalten) einer Relation auszuwählen. Die Projektion wird mit dem griechischen Buchstaben „Π“ (Π) bezeichnet.

Allgemein:

$R' := \Pi_{[R']}(R),$
 $[R'] \subseteq [R]$

Mit ...

$\Pi_{\text{Vorname}, \text{Nachname}}(M_S)$

..erhalten wir als Ergebnis nachfolgende Relation:

Vorname	Nachname
Hans	Müller
Rita	Schulze
Werner	Maier
Karin	Schwarz

Erinnern wir uns was wir beim Unterschied zwischen Relation und Tabelle in Kap.3.1.2 kennengelernt haben und stellen uns daher die Frage, was ist das Ergebnis folgender Projektion.

$\Pi_{\text{Skill}}(\text{M_S})$

Da es in einer Relation keine doppelten Tupel vorkommen dürfen, wird im Unterschied zu einer Abfrage in SQL, der Attributwert „PR“ nicht zweimal angezeigt.

SQL liefert als Ergebnis von Anfragen eine *Multimenge* zurück, also eine Menge, die Elemente mehrfach enthalten kann. Dies wurde aus Performance-Gründen so gehandhabt, um den zusätzlichen Schritt der Duplikatentfernung zu sparen.

Skill
PR
DBA
Test

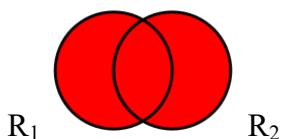
Natürlich lassen sich auch verschiedene Operationen miteinander verknüpfen. In der nachfolgenden Operation wird zuerst auf der Relation „Mitarbeiter_Softwarehaus“ die Selektion aller weiblichen Mitarbeiter (Geschlecht=‘w’) durchgeführt. Danach erfolgt eine Projektion auf das Attribut „Vorname“, sodass das Ergebnis der Verknüpfung, die Vornamen der weiblichen Mitarbeiterinnen, ist (Rita, Karin).

$\Pi_{\text{Vorname}}(\sigma_{\text{Geschlecht}='w'}(\text{M_S}))$

Ein Vertauschen der beiden Operationen ist jedoch nicht möglich, denn wenn zuerst eine Projektion auf das Attribut „Vorname“ stattfindet, kann die Selektion nicht mehr stattfinden, da das Attribut „Geschlecht“ in der Ergebnisrelation nicht mehr vorkommt.

6.4 Vereinigung (U)

In der Vereinigung werden zwei Relationen miteinander „geschnitten“.



Dies ist allerdings nur erlaubt, wenn die beiden Eingabeschemata der Ausgangsrelationen gleich sind.

Allgemein:

$$R' := R_1 \cup R_2$$

$$[R'] := [R_1] = [R_2]$$

D.h. wir bräuchten eine Relation mit einigen identischen Attributen welche wir derzeit jedoch nicht definiert haben. Wir führen daher eine neue Relation „Kunden“ ein.

Kunden

Kunden-Nr	Vorname	Nachname	Firma	Adresse
1	Manfred	Schwarz
2	Claudia	Müller
3	Klaus	Brecht
4	Martin	Klein

Nun verwenden wir die bereits bekannte Operation der Projektion auf die Tabelle Mitarbeiter_Softwarehaus und Kunden und Vereinigen diese beiden Ergebnisrelationen miteinander.

$$\Pi_{\text{Nachname}}(\text{M_S}) \cup \Pi_{\text{Nachname}}(\text{Kunden})$$

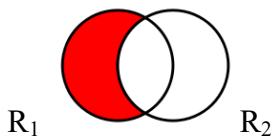
Wir erhalten dann folgende neue Relation:

Nachname
Müller
Schulze
Maier
Schwarz
Brecht
Klein

Wie wir sehen und wahrscheinlich aufgrund der Mengeneigenschaften auch bereits geahnt haben, sind die doppelten Namen in der Ergebnisrelation nicht mehr vorhanden.

6.5 Differenz (-)

Die nächste Mengenoperation ist die Differenz, gekennzeichnet mit einem Minuszeichen (auch „\“).



Dies ist allerdings nur erlaubt, wenn wieder die beiden Eingabeschemata der Ausgangsrelationen gleich sind.

Allgemein:

$$R' := R_1 - R_2$$

$$[R'] := [R_1] = [R_2]$$

Wir verwenden wieder die beiden Ergebnisrelationen der Projektion auf die Namen der Mitarbeiter_Softwarehaus und Kunden. Zum besseren Verständnis sind beide Ergebnisrelationen nochmals aufgeführt.

Mitarbeiter_Softwarehaus

Kunden

Nachname
Müller
Schulze
Maier
Schwarz

Nachname
Schwarz
Müller
Brecht
Klein

Mitarbeiter_Softwarehaus – Kunden

Nachname
Schulze
Maier

Die Relation „Kunde“ wird von der Relation „Mitarbeiter_Softwarehaus“ abgezogen und als Ergebnis erhält man nur noch die Namen „Schulze“ und „Maier“. Denn „Schwarz“ und „Müller“ kann man abziehen, d.h. diese fallen raus, „Brecht“ und „Klein“ sind in der ersten Relation nicht vorhanden, können also auch nicht abgezogen werden.

6.6 Kreuzprodukt (x)

Das Kreuzprodukt haben wir bereits im Kap. 3.1.1 kennengelernt, hier aber wird es auf zwei Relationen angewandt. Daher führen wir nun eine weitere Relation, nämlich die der „Vergütungsgruppe“ ein.

Relation: „Vergütungsgruppen“

Gehalts-Stufe	Kurzbeschreibung
It1	Expert 1
It2	Expert 2
It3	Expert 3

Allgemein:

$$R' := R_1 \times R_2$$

$$[R'] := [R_1] \cup [R_2]$$

Bilden wir nun das Kreuzprodukt zwischen den Relationen „Vergütungsgruppen“ (VG) und „Mitarbeiter_Softwarehaus“ (M_S) so ergibt sich die neue Relation aus der Kombination aller Attribute. Die beiden Relationschemata bilden das Schema der neuen Relation.

Mitarbeiter_Softwarehaus x Vergütungsgruppe

Pers-Nr	Vn	Nn	G	G_N	E_D	Sk	GS	VG.GS	Kurzbezeichnung
1	Hans	Müller	m	NULL	1.7.2001	PR	It2	It1	Expert 1
...
2	Rita	Schulze	w	NULL	1.11.2007	DBA	It3	It1	Expert 1
...
3	Werner	Maier	m	NULL	1.1.2010	Test	It2	It1	Expert 1
...
4	Karin	Schwarz	w	Klein	1.3.2005	PR	It2	It1	Expert 1
...
4	Karin	Schwarz	w	Klein	1.3.2005	PR	It2	It2	Expert 2
...

Diese Relation enthält nur alle möglichen Paare von Tupeln aus der Relation M_S und VG. Das Kreuzprodukt der beiden Relationen hat somit 10 Attribute. Haben zwei Attribute denselben Namen, so wird dies durch das Voranstellen des Relationennamens „VG.GS“ eindeutig.

Unsere beiden Relationen haben vier Tupel und drei Tupel und aus der Kombination ergeben sich dann $4 \times 3 = 12$ Tupel in der Ergebnisrelation.

Auch hier gilt wieder, dass mehrere Operationen miteinander kombiniert werden können und ist es möglich aus den 12 Tupel diejenigen extrahieren, bei denen die Gehaltsstufe aus beiden Relationen gleich ist, denn nur diese sind für uns eigentlich interessant.

$$\sigma_{M_S.GS = VG.GS} (M_S \times VG)$$

Wir erhalten dann als Ergebnisrelation die vier Tupel bei denen die Kurzbezeichnung zur Gehaltsstufe als Volltext ausgegeben wird (und eventuell noch weitere Angaben, welche aber hier nicht weiter ausgeführt sind).

6.7 Umbenennung (ρ)

Beim Kreuzprodukt hatten wir das Problem, dass wir in den beiden Relationen Attribute mit demselben Attributnamen haben. Soll im Prädikat der Operation nun wie oben beschrieben die beiden Attribute verwendet werden, so erfordert dies eine unübersichtliche Schreibweise. Dazu gibt es in der Relationenalgebra die Möglichkeit der Umbenennung von Attributen. Dies wird mit dem griechischen Buchstaben „ ρ “ (Rho) bezeichnet.

Allgemein:

$$R' := \rho_{A' \leftarrow A} R$$

$$A \in [R]$$

Beispiel:

$$\rho_{\text{neu} \leftarrow \text{alt}}(R)$$

$$\rho_{\text{Vergütungsstufe} \leftarrow \text{Gehaltsstufe}}(\text{Vergütungsgruppe})$$

Diese Operation ist wichtig um

- kartesische Produkte zu ermöglichen, bei denen es gleiche Attributnamen gibt, insbesondere auch beim kartesischen Produkt einer Relation mit sich selbst
- Mengenoperationen zwischen Relationen mit unterschiedlichen Attributen zu ermöglichen.

Auch der Relationenname kann mit dieser Operation umbenannt werden.

Allgemein:

$$R' := \rho_{R'}(R)$$

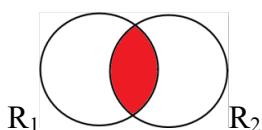
$$[R'] = [R]$$

Die Umbenennung einer Relation kann dann erforderlich werden, wenn wir in einer Abfrage eine Relation mehrfach verwenden möchten (siehe Joins).

Nun haben wir alle Basisoperatoren der Relationalen Algebra definiert. Alle nachfolgenden Operatoren können durch die Kombination mit diesen gebildet werden. Diese weiteren Operatoren sind im Prinzip kurz Schreibweisen von komplexeren zusammengesetzten Operatoren.

6.8 Durchschnitt (\cap)

Ganz ähnlich verhält es sich mit dem Durchschnitt zweier Relationen.



Auch hier ist diese Operation nur erlaubt, wenn die beiden Eingabeschemata der Ausgangsrelationen gleich sind.

Allgemein:

$$R' := R_1 \cap R_2$$

$$[R'] := [R_1] = [R_2]$$

Wir können aber nun mit unseren Basisoperatoren das Ganze auch so schreiben:

$$R' := R_1 - (R_1 - R_2)$$

Mitarbeiter
Softwarehaus

Nachname
Müller
Schulze
Maier
Schwarz

Kunden

Nachname
Schwarz
Müller
Brecht
Klein

Mitarbeiter _ Softwarehaus – Kunden

Ergebnisrelation

Nachname
Schulze
Maier

Mitarbeiter _ Softwarehaus – *Ergebnisrelation*

Nachname
Müller
Schulze
Maier
Schwarz

Nachname
Schulze
Maier

Nachname
Müller
Schwarz

Entspricht:

Mitarbeiter _ Softwarehaus \cap Kunden

6.9 Theta-Join (\bowtie_p)

Der Theta-Join ist im Prinzip das Kreuzprodukt zweier Relationen und darauf eine Selektion durchgeführt.

Allgemein:

$$TJ := R_1 \bowtie_p R_2$$

$$[TJ] := [R_1] \cup [R_2]$$

Mit den Basisoperatoren geschrieben lautet der Theta-Join:

$$R' := \sigma_p(R_1 \times R_2) \subseteq (R_1 \times R_2)$$

wobei das Prädikat „P“ eine Vereinigung der beiden Schemata der Ausgangsrelation ist.

$$P: [R_1] \cup [R_2]$$

und hierbei handelt es sich wieder um einen booleschen Ausdruck.

Wenn wir uns noch an unser Kreuzprodukt aus Kap. 6.6 erinnern, hier haben wir sehr viele Tupel erhalten, welche uns im Prinzip nicht interessierten. Was uns aber in diesem Kreuzprodukt durchaus interessiert sind die Tupel bei denen die Gehaltsstufe der Relation Mitarbeiter_Softwarehaus gleich der der Relation Vergütungsstufe (haben wir in Kap. 6.7 umbenannt) sind. So erhalten wir zu jeder Gehaltsstufe unserer Mitarbeiter die zugehörige Kurzbezeichnung. Wir erinnern uns noch.

$$\sigma_{GS=VS}(M_S \times VG)$$

dies lässt sich nun auch mit dem Theta-Join darstellen. Zusätzlich können wir noch eine weitere Selektion im Prädikat einführen.

Mitarbeiter_Softwarehaus $\bowtie_{GS=VS \wedge GS='It3'} VG$

Beim ersten Vergleich im Prädikat erhalten wir die nachfolgende Tabelle, bei der mit der über da UND verknüpften Selektion nur noch die Rita Schulze übrig bleibt.

Pers-Nr	Vn	Nn	G	G_N	E_D	Sk	GS	VS	Kurzbezeichnung
1	Hans	Müller	m	NULL	1.7.2001	PR	It2	It2	Expert 2
2	Rita	Schulze	w	NULL	1.11.2007	DBA	It3	It3	Expert 3
3	Werner	Maier	m	NULL	1.1.2010	Test	It2	It2	Expert 2
4	Karin	Schwarz	w	Klein	1.3.2005	PR	It2	It2	Expert 2

Wie wir an der Ergebnisrelation feststellen können, werden die Attribute „Gehaltsstufe (GS)“ und „Vergütungsstufe (VS)“ aus beiden Relationen angezeigt.

Dieser Verbund ist, wenn wir mit Relationalen Datenbanken arbeiten, der häufigste vorkommende Verbund und dort sprechen wir dann auch nur vom „Join“. Es gibt nun noch verschiedene Joins - wir wollen nachfolgend nur noch zwei kurz behandeln - die alle auf dem Theta-Join aufbauen.

6.10 Equi-Join ($\bowtie_{[L],[R]}$)

Der Equi-Join ist wiederum eine Sonderform des Theta-Joins bei dem ein Teilschema der linken Relation vor dem Jionsymbol gleich einem Teilschema der rechten Relation sein muss.

EJ:= $R_1 \bowtie_{[L],[R]} R_2$

Entspricht dem Theta-Join

= $R_1 \bowtie_P R_2$

mit

$P:=R_1.[L] = R_2.[R]$ und $[L] \subseteq R_1, [R] \subseteq R_2$

Die bedeutet für unsere Relation, dass wenn wir folgende Operation ausführen

Mitarbeiter_Softwarehaus $\bowtie_{GS,VS}$ Vergütungsgruppen

Wir erhalten wieder die Ergebnisrelation mit den vier Tupel.

6.11 Natürliche Join (\bowtie)

Es gibt auch noch den „Natürlichen Join“ der wiederum auf dem Equi-Join aufbaut. Dieser natürliche Verbund wird ohne Prädikat geschrieben.

NJ:= $R_1 \bowtie R_2$

$\Pi_{[NJ]}(R_1 \bowtie_{[J],[J]} R_2)$

mit

$[J]:=[R_1.] \cap [R_2]$ und $[NJ]:= [R_1] \cup ([R_2]-[J])$

Dies bedeutet, etwas einfacher ausgedrückt, dass wir die beiden Attribute der beiden Relationen miteinander verbinden, welche denselben Namen aufweisen. Da dies zwingend ist, kann es zunächst erforderlich sein, eine Umbenennung der Attribute vorzunehmen, falls die Attribute, welche wir verwenden möchten, unterschiedliche Namen aufweisen. Bei diesem Join wird in der Ergebnisrelation das Verbundattribut nur einmal angezeigt. Es qualifizieren sich wieder die Tupel, welche dieselben Attributwerte aufweisen.

Würden wir wieder unsere beiden Relationen Mitarbeiter_Softwarehaus und Vergütungsstufen miteinander verbinden, dann müssten wir das Attribut „Vergütungsstufe“ wieder in „Gehaltsstufe“ umbenennen, denn sonst funktioniert dieser Join nicht.

Mitarbeiter_Softwarehaus \bowtie Vergütungsgruppen

Pers-Nr	Vn	Nn	G	G_N	E_D	Sk	GS	Kurzbezeichnung
1	Hans	Müller	m	NULL	1.7.2001	PR	It2	Expert 2
2	Rita	Schulze	w	NULL	1.11.2007	DBA	It3	Expert 3
3	Werner	Maier	m	NULL	1.1.2010	Test	It2	Expert 2
4	Karin	Schwarz	w	Klein	1.3.2005	PR	It2	Expert 2

Weitere Join-Operatoren

Es gibt noch eine Reihe weiterer Joins auf die wir hier nicht näher eingehen werden. Zur Vertiefung empfehle ich ([KeEi15] ,S.95-97) oder ([ElNa05] ,S.166-168).

Die in Kap. 6.9 bis Kap.6.11 beschrieben Joins enthalten in der Ergebnisrelation keine Tupel welche keinen Joinpartner gefunden haben. Man nennt sie daher auch „Innere Joins“. Bei den „Äußeren Joins“ welche wir hier nicht behandeln, werden auch solche Tupel aus der linken oder rechten Relation angezeigt (je nach Art des Joins) welche beim Join „partnerlos“ geblieben sind.

Auch auf die Gruppierung und Aggregation werden wir nur im Kap. SQL eingehen.

7 SQL Teil 2

7.1 Daten abfragen (DRL)

Kommen wir nun zum wichtigsten Teil innerhalb der Sprache SQL, nämlich der Abfrage von Daten aus einer Datenbank. Dieser Teil ist so mächtig, dass es uns nicht möglich sein wird alle Varianten der Data Retrieval Language (DRL, oder zu Deutsch Abfragesprache) zu behandeln. Wir werden uns daher auf die wichtigsten beschränken und ich möchte es den Studierenden überlassen sich anhand der umfangreichen Dokumentation von PostgreSQL mit den weiteren Möglichkeiten zu beschäftigen.

7.1.1 Einfache Abfrage von Daten

Die allgemeine Struktur eines SELECT Statements besteht aus den drei Teilen

SELECT, FROM und WHERE:

```
SELECT Attribut1, Attribut2, ..., Attributn
FROM Relation1, Relation2, ..., Relation_n
WHERE Prädikat
```

1. Der SELECT-Teil entspricht der Projektionsoperation (Π) der relationalen Algebra. Er wird verwendet, um diejenigen Attribute aufzulisten, die im Ergebnis einer Abfrage gewünscht werden.
2. Der FROM-Teil entspricht der Bildung eines kartesischen Produkts (X) in der relationalen Algebra. Ist nur eine Relation angegeben, wird nur diese als Eingabe verwendet.
3. Der WHERE-Teil entspricht dem Selektionsprädikat (σ) der relationalen Algebra. Er besteht aus einem Prädikat (Bedingung), in dem Attribute derjenigen Relationen auftreten, die in der FROM-Klausel vorkommen. Diese Bedingung wird auf das Ergebnis aus dem FROM-Teil angewendet. Dieser Teil ist optional.

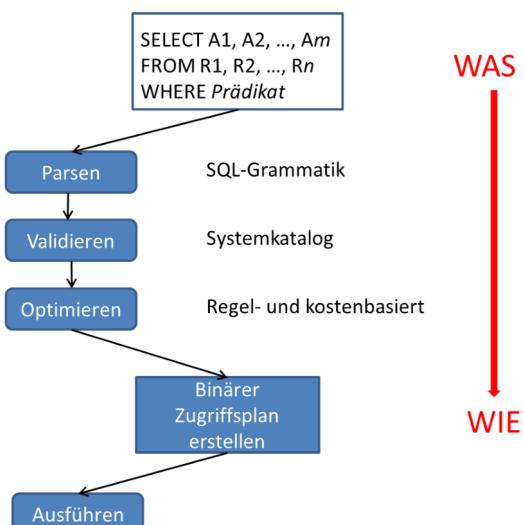


Abbildung 37: Ausführung SQL-Statement SELECT

Nun wollen wir uns anschauen, wie das DBMS das nachfolgende konkrete Beispiel abarbeitet.

```
SELECT *
FROM Kunde, Auftrag;
```

Was passiert nun genau in der FROM-Klausel?

Dazu schauen wir uns als Erstes das Kreuzprodukt der Tabellen *Kunde* und *Auftrag* an. *Kunde* hat in unserem Beispiel 5 Tupel und *Auftrag* hat ebenfalls 5 Tupel. Dies ergibt im Ergebnis 25 Tupel mit den Spalten (Attributten) aller Ausgangsrelationen. Das Kreuzprodukt haben wir bereits ausführlich in Kap. 6.6 behandelt.

In der SELECT-Klausel wird nur die Auswahl der Spalten aus der Ergebnisrelation getroffen. Der Stern in Selektion sagt, dass alle Spalten aus der Ergebnisrelation übernommen werden.

Würden wir noch eine WHERE-Klausel hinzufügen, so wird dann das Ergebnis noch eingeschränkt indem die gesetzte Bedingung (Prädikat) erfüllt wird.

```
SELECT *
FROM Kunde
WHERE Nachname = 'Schwarz'
; 
```

Als Ergebnis erhält man nur noch das Tupel des Kunden „Schwarz“.

An dieser Stelle sei noch bemerkt, dass bei Textvergleichen in der WHERE-Klausel die Groß-/Kleinschreibung zu beachten ist.

Als Vergleichsoperatoren sind natürlich nicht nur das Gleichheitszeichen zulässig, sondern alle nachfolgenden. Diese können nicht nur auf numerische Werte, sondern auch auf Datentypen wie Character, Datum, Uhrzeit usw. angewandt werden.

Operatoren: `=, <>, <, >, <=, >=`

Hinweis:

Die Reihenfolge für die Abarbeitung des SELECT ist:

1. Zuerst wird das FROM ausgeführt
2. dann das WHERE durchgeführt
und für alle Tupel welche die Bedingung überleben wird
3. das SELECT als Projektion durchgeführt.

7.1.1.1 Distinct und Order By

Wie bereits im Kapitel der Relationalen Algebra erwähnt, hat eine Relation keine doppelten Tupel, in SQL aber sehr wohl. Dies bedeutet, dass wenn wir bei einer Abfrage die doppelten Werte ausschließen wollen, müssen wir dies explizit angeben. Dies tun wir mit der Anweisung DISTINCT.

```
SELECT DISTINCT Vorname
FROM Mitarbeiter
; 
```

Wie ebenfalls in früheren Kapiteln bereits erwähnt, ist die Sortierreihenfolge in einer Tabelle nicht vorgegeben und völlig beliebig. Soll die Ausgabe in eine bestimmten Reihenfolge erfolgen, so verwenden wir hier den Zusatz ORDER BY.

```
SELECT DISTINCT Vorname
FROM Mitarbeiter
ORDER BY Vorname
; 
```

Wollen wir noch auf- oder absteigend sortieren, so sagen wir dies dem DMBS mit ASC (Ascending) oder DESC (Descending) beim ORDER BY.

7.1.1.2 Wertelisten und flexibles Suchen

Die Anforderungen an das Suchen in den Daten einer Datenbank sind mannigfaltig, daher werden wir uns nachfolgend noch mit zwei Möglichkeiten beschäftigen.

Wollen wir beispielsweise Tupel suchen, bei denen ein Wert einer aus einer Liste von Werten (Wertebereich) enthalten ist, so erreichen wir dies mit folgender Anweisung.

```
SELECT * from Mitarbeiter
WHERE Nachname in ('Schulze', 'Kunze', 'Brecht')
;
```

Die Umkehrung dazu lautet NOT IN.

Mit BETWEEN und AND können wir bei der Abfrage einen Wertbereich betrachten (NOT ist auch möglich).

```
... WHERE Eintrittsdatum Between '01.01.2010' and '01.01.2015'
```

Um nun die Suche in Feldern mit dem Datentyp „Character“ noch flexibler zu ermöglichen, gibt es den Operator LIKE. Mit diesem kann nach bestimmten Textmustern gesucht werden. Hierzu gibt es zwei Zeichen („_“, „%“), um ein Zeichen an einer bestimmten Position zu ersetzen, oder es darf ein unbestimmtes beliebiges Zeichen in der Spalte auftreten.

```
... WHERE Nachname Like '%cht'
```

Mit dieser Abfrage würden wir die Zeilen der Mitarbeiter „Brecht“ und „Habrecht“ angezeigt bekommen.

7.1.1.3 Funktionen

In SQL ist eine große Anzahl von Funktionen innerhalb des SELECT-Befehls möglich. Nachfolgende wollen wir uns einige wenige genauer anschauen.

Wenn wir das Durchschnittsgehalt aller Mitarbeiter unseres Softwarehauses wissen wollen müssen wir schreiben:

```
SELECT AVG (Gehalt)
from Mitarbeiter
;
```

Hier würden wir dann einen Ergebniswert für das Durchschnittsgehalt erhalten.

Fragen wir uns aber wie groß die Summe der noch nicht bearbeiteten Leistungen (aus Aufträgen) ist, so müssen wir für alle Leistungen bei denen noch kein Ende Datum eingetragen ist, die geplanten Projektstunden aufaddieren.

```
SELECT SUM (gepl_std)
from Leistung
Where Ende_termin is null
;
```

Nun können wir noch die Anzahl der Mitarbeiter zählen (wissen wir natürlich)

```
SELECT count (*) from Mitarbeiter;
```

oder aber ausgeben welches das kleinste und größte Gehalt im Softwarehaus ist.

```
SELECT Min (Gehalt), Max (Gehalt)
from Mitarbeiter
;
```

Lassen wir es damit bewenden, denn die Anzahl der Möglichkeiten würde unseren Rahmen bei weitem sprengen.

7.1.1.4 Gruppenbildung

Im vorherigen Kapitel haben wir mit COUNT (*) die Anzahl der Mitarbeiter gezählt und dies bedeutet, dass das DBMS aus allen Mitarbeitern eine Gruppe bildet und dann innerhalb dieser Gruppe die Anzahl berechnet. Diese Funktionen liefern aber immer nur ein Ergebnis, nun wollen wir aber auch Ergebnisse über mehrere Spalten hinweg ermitteln. Hierzu dient uns die GROUP BY-Klausel. Das Gruppenkriterium kann aus einer oder mehrerer Spalten bestehen.

Interessiert uns nun wie viele Mitarbeiter in den einzelnen Skill-Gruppen (PR, Admin..) zugeordnet sind und was das Durchschnittsgehalt innerhalb dieser Gruppen ist, so müssen wir zuerst gruppieren und dann die erwünschten Berechnungen durchführen.

```
SELECT Skill, Count (*), AVG (Gehalt)
from Mitarbeiter
Group By Skill
Order By Skill
;
```

	skill character varying (5)	count bigint	avg numeric
1	Analy	2	64000.00
2	DBA	1	61000.00
3	PL	1	65000.00
4	PR	1	56000.00
5	Test	2	43000.00
6	[null]	3	41666.66

Abbildung 38: Ergebnis des GROUP BY

Alle Zeilengruppen, die vom DBMS auf unsere Anweisung hin gebildet wurden, liefern jeweils eine Ergebnisspalten. Als Ergebnisspalten dürfen allerdings nur die Attribute verwendet werden, welche auch in der GROUP BY Klausel angegeben sind.

Nun gibt es in der Praxis auch Anforderungen, bei denen die Ergebniswerte der Gruppierungen eingeschränkt werden sollen. Z.B. möchte ich alle Abteilungen angezeigt bekommen, bei denen das Durchschnittsgehalt > 50000€ beträgt. In diesem Fall müssten wir nach Abteilungen gruppieren und dann mit der HAVING-Klausel die Ergebnisse noch einschränken.

```
SELECT arbeitet_in, Count (*), AVG (Gehalt)
from Mitarbeiter
Group By arbeitet_in
having avg (Gehalt) > 50000
;
```

Da unser Softwarehaus aus drei Abteilungen besteht, aber das Durchschnittsgehalt der Mitarbeiter in einer Abteilung (EDM2) < 50000€ ist, wird diese im Ergebnis nicht angezeigt.

	arbeitet_in character varying (8)	count bigint	avg numeric
1	EDM3	3	61333
2	EDM1	3	51333

Auch in der HAVING Klausel kann nur auf die Werte eingeschränkt werden, welche in den Zeilengruppen gleich sind.

7.1.1.5 Zusammenfassung des SELECT

Wir haben nun in den vorangegangenen Kapiteln die komplette SELECT-Anweisung kennengelernt und wollen nun diese in diesem Kapitel nochmals kurz zusammenfassen.

```
SELECT ...
from ...
Where ...
Group By ...
Having ...
Order By ...
;
```

Zwingend notwendig ist beim SELECT nur die Angabe FROM, alle anderen Klauseln sind optional.

In Kap. 7.1.1 haben wir bereits die Abarbeitungsreihenfolge des SELECTs kurz angesprochen, wir wollen diese aber für die gesamten Klauseln kurz ergänzen.

1. Zuerst wird das FROM ausgeführt und eventuell ein Kreuzprodukt gebildet. Hier sind noch die Spalten aller beteiligten Tabellen vorhanden.
2. Nun werden über die WHERE Klausel alle Zeilen aus dieser „Zwischentabelle“ entfernt, welche der Suchbedingung nicht entsprechen.
3. Über die GROUP BY Klausel werden nun Zeilengruppen gebildet, welche die Anzahl der Zeilen aber noch nicht verringern.
4. Durch die HAVING Bedingung werden nun alle Zeilengruppen entfernt, welche die Bedingung nicht erfüllen.
5. Im SELECT werden dann die Zeilen innerhalb der Gruppe zu einer Zeile zusammengefasst und durch die Projektion nur noch die gewünschten Spalten ins Ergebnis übernommen.
6. Zum Schluss werden die Zeilen nach der Angabe in der ORDER BY-Klausel sortiert und ausgegeben.

Zum Abschluss noch ein vollständiges Beispiel.

Wir wollen nun das Durchschnittsgehalt der Mitarbeiter wissen, welche nach dem 1.1.2014 im Softwarehaus eingestellt wurden, bezogen auf die Skill-Gruppen (NULL ist keine Skill-Gruppe) und alles nach Durchschnittsgehalt sortiert.

```
SELECT Skill, AVG (Gehalt)
from Mitarbeiter
Where Eintrittsdatum >= '1.1.2014'
Group By Skill
having Skill is not NULL
Order By 2
;
```

7.1.2 Unterabfragen

Wollen wir nun wissen, welche Mitarbeiter in unserer Firma über dem Durchschnittsgehalt aller Mitarbeiter verdienen, so müssen wir dies mit dem derzeitigen Wissen in zwei Stufen abhandeln.

1. Select Avg(Gehalt) from Mitarbeiter;
2. Select Vorname, Nachname, Gehalt
from Mitarbeiter
Where Gehalt > *Ergebnis aus 1*
Order By Gehalt;

Nun gibt es in der WHERE Klausel aber nicht nur die Möglichkeit mit einem festen Wert zu vergleichen, sondern auch eine variable Abfrage mit SELECT einzubauen.

Dies nennt man eine Unterabfrage oder subquery.

Anstatt der Abfrage mit den zwei SELECTs, lässt sich mit einer Unterabfrage viel eleganter lösen.

```
Select Vorname, Nachname, Gehalt
from Mitarbeiter
Where Gehalt > (Select Avg(Gehalt) from Mitarbeiter)
Order By Gehalt;
```

Man spricht hier nun von einer Hauptabfrage und einer Unterabfrage, welche durch Klammern getrennt sind. Unterabfragen sind auch in der HAVING-Klausel möglich.

Bei der Unterabfrage ist zu beachten, dass das Ergebnis einem Vergleichsoperator entspricht, denn normalerweise ist das Ergebnis eines SELECTs wieder eine Tabelle und dies ist nicht zulässig. Veranschaulich wir dies kurz in einem Beispiel.

```
...
Where Gehalt >
(Select Gehalt from Mitarbeiter
where Pers_Nr=1)...
```

Diese Unterabfrage wäre zulässig, da sie nur einen Wert liefert. Wenn wir aber nur eine Kleinigkeit ändern und schreiben:

```
...
Where Gehalt >
(Select Gehalt from Mitarbeiter
where Pers_Nr>1)...
```

ist dies nicht mehr zulässig, da wir mehr als einen Ergebniswert aus der Unterabfrage erhalten.

Hierbei würde ein Ergebniswert (Gehalt) mit dem Inhalt einer Tabelle verglichen werden und dies ist natürlich nicht möglich.

Die Unterabfrage ist so zu formulieren, dass deren Attributliste aus nur einem Element besteht (Attributname, Funktion oder arithmetischer Ausdruck). Des Weiteren muss die WHERE-Klausel in der Unterabfrage so formuliert werden, dass diese nur eine Ergebniszeile zurück liefert.

Nun wären mit dieser Einschränkung viele Abfragen nicht möglich, daher bietet SQL auch eine Möglichkeit Unterabfragen mit mehreren Ergebniszeilen zu verarbeiten. Dazu muss der Unterabfrage einer der nachfolgenden Ausdrücke vorangestellt werden.

ANY, ALL, EXISTS, IN ...

Exemplarisch wollen wir den Ausdruck ANY kurz behandeln.

Folgende Anfrage soll uns als Beispiel dienen.

Wir suchen alle Mitarbeiter und deren Skill am Standort „Leinfelden“. Ist kein Skill angegeben (NULL) wird der Mitarbeiter nicht ausgegeben. Wir sortieren nach Nachname (3. Zeile-1).

```

Select Vorname, Nachname, Skill, arbeitet_in
From Mitarbeiter
Where arbeitet_in =
Any (Select Abt_bez_kurz
      from Abteilung
      Where Standort ='Leinfelden')
      and
      Skill is not NULL
      Order by 2
;
  
```

Wie verarbeitet das DBMS nun diese Anfrage?

Die Unterabfrage, welche auch nur ein Attribut im Ergebniswert enthalten darf, liefert eine Anzahl Zeilen zurück. Der Ergebniswert jeder Zeile wird nun mit dem linken Wert der WHERE-Klausel verglichen. Das Ergebnis des Vergleichs ist „wahr“(true), wenn mindestens ein Ergebniswert der Unterabfrage die Vergleichsbedingung erfüllt. Erfüllt kein Ergebniswert die Bedingung, dann liefert der Vergleich „unwahr“(false) zurück.

Unterabfragen beim INSERT

Natürlich sind Unterabfragen nicht nur beim SELECT möglich, sondern auch in anderen SQL-Befehlen der DML, wie beispielsweise beim INSERT.

Wir gehen nun kurz ein paar Schritte zurück und wenden uns nochmals dem CREATE-Statement zu. Hier hatten wir einen wichtigen Datentyp bewusst nur am Rande kurz erwähnt und die Beschreibung dieses Datentyps (SERIAL) wollen wir an dieser Stelle, im Zusammenhang mit der Unterabfrage beim INSERT, nachholen.

Wollten wir bisher einen Datensatz z.B. in die Tabelle *Mitarbeiter* einfügen, so mussten wir wissen, welches die höchste Personalnummer ist, um diese dann beim Einfügen des neuen Datensatzes um einen Wert zu erhöhen. Mit der zuvor behandelten Unterabfrage ist es nun möglich, den höchsten Wert aus der Tabelle auszulesen und diesen beim INSERT direkt, um einen Wert erhöht, einzufügen.

```

Insert into Mitarbeiter
Values (
  (Select max(Pers_nr)+1 From Mitarbeiter),
  'Horst', 'Mahler', NULL, '01.04.1975', 'm', '15.3.2013',
  default, 49000)
; 
```

Dieses Select max(Pers_nr)+1 From Mitarbeiter können wir weglassen, wenn wir bereits beim Anlegen der Tabelle (oder später durch ALTER) das sogenannte Autoincrement mit dem Constraint SERIAL setzen. Bei der Tabelle *Mitarbeiter* würde der Befehl dann folgendermaßen lauten:

```

CREATE TABLE Mitarbeiter
(
  Pers_Nr Serial Primary Key,
  Vorname varchar,
  ....
  
```

Wir verlassen nun den Bereich der Abfragen einzelner Tabelle und wenden uns dem Teil der DQL zu, welche bei einem komplexen ERM sehr häufig benötigt wird:
 Der Abfrage mehrerer Tabellen.

7.1.3 Mehrere Abfragen miteinander verknüpfen

In der Relationalen Algebra haben wir außer dem Kreuzprodukt noch weitere Möglichkeiten kennengelernt, wie man Tabellen miteinander verknüpft. Dies waren die Vereinigung, der Durchschnitt und die Differenz (siehe Kap. 6.4 bis 6.8). Nachfolgend wollen wir uns die SQL-Anweisung dazu anschauen.

7.1.3.1 UNION (Vereinigung)

Mit UNION oder auf Deutsch, Vereinigung (\cup) werden die Ergebnismengen zweier Abfragen miteinander vereinigt.

```
Select Vorname
from Mitarbeiter
UNION
Select Vorname
from Kunde
;
```

In der vorherigen Abfrage erfolgt eine Projektion auf Vorname der Tabelle Mitarbeiter und diese wird „geschnitten“ mit der Projektion auf Vorname der Tabelle Kunde. Diese liefert uns alle Vornamen der beiden Tabellen welche nicht doppelt vorhanden sind. Dies haben wir bereits in der Relationalen Algebra so gezeigt (siehe Kap. 6.4).

Wollen wir die Duplikateliminierung ausschalten und alle Vornamen der beiden Tabellen angezeigt bekommen, so schreiben wir UNION ALL.

7.1.3.2 INTERSECT (Durchschnitt)

Der Durchschnitt (\cap), englisch INTERSECT gibt an, welche Attribute zweier Abfragen im Schnitt der beiden (Mengen) liegen. Nehmen wir wieder die obige Abfrage, so enthält das Ergebnis eine Liste von Vornamen, welche in beiden Tabellen vorhanden sind.

```
Select Vorname
from Mitarbeiter
INTERSECT
Select Vorname
from Kunde
;
```

Beim INTERSECT ALL werden wieder alle Werte angezeigt, welche in **beiden** Ausgangstabellen mehrfach vorkommen.

7.1.3.3 EXCEPT (Differenz)

Die Differenz (Minus) zweier Mengen oder Abfragen wird in SQL EXCEPT genannt. Falls die genaue Funktion nicht mehr ganz geläufig ist, soll auf Kap. 6.5 verwiesen werden. Im nachfolgenden, bereits bekannten Beispiel, ziehen wir nun die Vornamen beider Spalten voneinander ab. Überlegen Sie kurz, was Ihrer Meinung nach das Ergebnis sein wird.

```
Select Vorname
from Mitarbeiter
EXCEPT
Select Vorname
from Kunde
;
```

Die Vornamen, welche in der Tabelle Kunde und in der Tabelle Mitarbeiter vorhanden sind, werden voneinander abgezogen, fallen also raus und das was übrig bleibt, wird ohne doppelte Werte angezeigt. Auch hier wird wieder mit EXCEPT ALL die Duplikateliminierung ausgeschaltet.

7.1.4 Abfragen auf mehrere Tabellen

Bisher haben wir uns bei den Abfragen meistens auf eine Tabelle beschränkt. Im ERM unseres Softwarehauses haben wir aber bereits die Tabellen miteinander verknüpft und diese Verknüpfung dann auch in der DDL hergestellt. Nun wollen wir davon Gebrauch machen und entlang dieser Verknüpfungen die Tabellen in den Abfragen miteinander verbinden. Auch hier werden wir uns eng an die Beispiele aus der Relationalen Algebra aus Kap. 6.9 halten.

7.1.4.1 Verknüpfung zweier Tabellen

Die einfachste Verknüpfung haben wir bereits bei der SELECT-Anweisung (Kap. 7.1.1) verwendet. Dies war das Kreuzprodukt zweier Tabellen. Hierbei wird jedes Tupel der einen Tabelle mit jedem Tupel der anderen Tabelle verknüpft. Das Schema der neuen Tabelle ist die Vereinigung der Schemata beider Ausgangstabellen.

Bilden wir das Kreuzprodukt von *Kunde* und *Auftrag* so erhalten wir folgendes Schema und 25 Tupel des Kreuzprodukts als Ergebnismenge.

kunden_nr	vorname	nachname	firma	intern	auftrag_nr	auftrags_datum	bezeichnung	abrechnungsart	erteilt_von
-----------	---------	----------	-------	--------	------------	----------------	-------------	----------------	-------------

Abbildung 39: Ergebnis-Schema von Kunde und Auftrag

Wir könnten nun mit einem SELECT die Tupel auswählen, bei dem die *Kunden_Nr* aus der Tabelle *Kunde* gleich dem Wert in Spalte *erteilt_von* aus der Tabelle *Auftrag* ist und wir würden alle Kunden und ihre zugehörenden Aufträge erhalten.

$\sigma_{\text{Kunden_Nr} = \text{erteilt_von}} (\text{Kunde} \times \text{Auftrag})$

```
Select *
from Kunde, Auftrag
Where Kunden_Nr=erteilt_von
;
```

Der schon mehrfach erwähnte Nachteil einer solchen Verbindung zweier Tabellen ist die immense Aufblähung durch das Kartesische Produkt.

7.1.4.2 Der INNERE JOIN

Theta-Join:

Mit SQL-92 wurde die Möglichkeit des JOINS geschaffen. Das vorige Select-Statement können wir nun mit einem Join-Operator folgendermaßen schreiben:

```
Select *
from Kunde JOIN Auftrag
On Kunden_Nr = erteilt_von
;
```

Was ist nun der Unterschied der beiden Statements? Was beim Kreuzprodukt (engl. Cross-Join) gemacht wird haben wir hinlänglich erwähnt. Was der Optimizer unseres DBMS letztendlich wirklich daraus macht und wie er dieses Statement abarbeitet, entzieht sich derzeit unserer Kenntnis. Wir müssen davon ausgehen, dass er ein Kartesisches Produkt erstellt. Durch den `JOIN` Operator aber sagen wir dem Optimizer, dass eine Fremdschlüsselbeziehung existiert, an derer entlang er die beiden Tabellen verbinden soll. Auch wie er dies macht ist ihm überlassen.

Diesen `JOIN` haben wir in Kap. 6.9 als Theta-Join beschrieben. Formal schreiben wir:

$TJ := R_1 \bowtie_p R_2$
 $[TJ] := [R_1] \cup [R_2]$

Kunde \bowtie Kunden_Nr = erteilt_von Auftrag

Das (Join-)Prädikat entspricht dem `ON...` beim `JOIN`. Hier kann ein beliebiges Prädikat stehen. Nun wollen wir auch gleich noch die anderen Join-Varianten aus Kap. 6.1 behandeln.

Equi-Join:

Der Equi-Join verbindet die Tabellen über die Werte zweier Tabellen, bei denen ein Teilschema der linken Relation vor dem Joinsymbol gleich einem Teilschema der rechten Relation sein muss.

$EJ := R_1 \bowtie_{[L],[R]} R_2$

Entspricht dem Theta-Join

$= R_1 \bowtie_p R_2$

Wir suchen nun die Namen aller Projektmitarbeiter und die Aufträge an denen Sie gearbeitet haben oder aktuell noch arbeiten.

Dazu müssen wir zuerst die Tabelle *arbeitet_an* mit der Tabelle *Aufträge* über die *Auftragsnummer* verbinden. Um nun die Namen der Projektmitarbeiter zu erhalten, müssen wir noch die Tabelle *Personen* dazu nehmen (joinen) und diese über die Projektnummer verbinden.

```
Select distinct Nachname, Vorname, Auftrag_nr
from
Auftrag JOIN arbeitet_an using (Auftrag_nr)
Join Mitarbeiter using (Pers_Nr)
order by nachname
;
```

Natural-Join:

Beim Natural-Join (als Spezialfall des Equi-Joins) werden zwei Tabellen miteinander verknüpft, welche denselben Namen und Inhalt haben.

$NJ := R_1 \bowtie R_2$

$\Pi_{[NJ]}(R_1 \bowtie_{[J],[J]} R_2)$

Als Beispiel nehmen wir hier die Tabelle *Mitarbeiter* und *Projektleiter*. Wir erinnern uns, dass die Projektleiter eine Spezialisierung der Mitarbeiter waren und daher nur noch die Attribute dieser

Spezialisierung enthalten. Die Verbindung (Fremdschlüssel) erfolgt über die Personalnummer, welche in beiden Tabellen den gleichen Namen aufweist.

```
Select *
from Mitarbeiter Natural JOIN Projektleiter
;
```

Mit dieser Anweisung erhalten wir die kompletten Daten der Mitarbeiter die auch Projektleiter sind.

Dies Joins, welche jeweils einen Join-Partner haben, werden auch oder `INNER JOIN` genannt. Dies kann man auch explizit beim `JOIN` schreiben, braucht es aber nicht, da dies per Default eingestellt ist.

```
Select *
from Mitarbeiter Natural Inner JOIN Projektleiter
;
```

7.1.4.3 Der ÄUSSERE JOIN

Wo es einen Inneren Join gibt, gibt es natürlich auch einen Äußenen Join (`OUTER JOIN`). Hier werden jeweils die Tupel mit angezeigt für die es keinen Join-Partner gibt. Zur Veranschaulichung wollen wir uns dazu zuerst ein Beispiel anschauen, bevor wir noch etwas tiefer in diese Joins einsteigen.

Wir wollen wissen, welche unserer Projekt-Mitarbeiter bisher noch an keiner Leistung gearbeitet haben. Um dies herauszufinden, müssen wir zuerst die Tabelle `arbeitet_an` mit der Tabelle `Mitarbeiter_Projekt` verbinden. Hier sagen wir nun, dass wir auch die Mitarbeiter suchen, für die es in `arbeitet_an` keinen Join-Partner gibt, die also noch an keiner Leistungen arbeiten (gearbeitet haben). Diese verbinden wir dann anschließend mit der Tabelle `Mitarbeiter` um über die Personalnummer den Namen des Mitarbeiters zu erhalten.

```
Select Pers_nr, Vorname, Nachname, Projekterfahrung
from arbeitet_an natural right outer join mitarbeiter_projekt
natural Join Mitarbeiter
where Leistung_nr is Null
;
```

Mit diesem Statement erhalten wir nun alle Projektmitarbeiter mit ihrer Projekterfahrung und auch einen Mitarbeiter, bei dem keine Leistungsnummer (NULL) erscheint.

Um die Unterschiede der einzelnen Joins nochmals zu verdeutlichen, greifen wir auf eine Abbildung aus [KeEi15] S.96 zurück.

Natürlicher Join

L			R			Resultat				
A	B	C	C	D	E	A	B	C	D	E
a ₁	b ₁	c ₁	c ₁	d ₁	e ₁	a ₁	b ₁	c ₁	d ₁	e ₁
a ₂	b ₂	c ₂	c ₃	d ₂	e ₂					

Linker äußerer Join

L			R			Resultat				
A	B	C	C	D	E	A	B	C	D	E
a ₁	b ₁	c ₁	c ₁	d ₁	e ₁	a ₁	b ₁	c ₁	d ₁	e ₁
a ₂	b ₂	c ₂	c ₃	d ₂	e ₂	a ₂	b ₂	c ₂	-	-

Rechter äußerer Join

L			R		Resultat					
A	B	C	C	D	E	A	B	C	D	E
a ₁	b ₁	c ₁	c ₁	d ₁	e ₁	a ₁	b ₁	c ₁	d ₁	e ₁
a ₂	b ₂	c ₂	c ₃	d ₂	e ₂	-	-	c ₃	d ₂	e ₂

Äußerer Join (voller)

L			R		Resultat					
A	B	C	C	D	E	A	B	C	D	E
a ₁	b ₁	c ₁	c ₁	d ₁	e ₁	a ₁	b ₁	c ₁	d ₁	e ₁
a ₂	b ₂	c ₂	c ₃	d ₂	e ₂	a ₂	b ₂	c ₂	-	-
-	-	-	-	-	-	-	-	c ₃	d ₂	e ₂

Abbildung 40: Ergebnisse verschiedener Joins
 (siehe [KeEi15] S.96)

7.2 Views in SQL

7.2.1 Einfache Sichten

Wir haben im Kap. 3.2.3 die Umsetzung der Spezialisierung der Mitarbeiter des Softwarehauses in verschiedenen Tabellen beschrieben. Die Tabellen der Spezialisierung wie beispielsweise die der Projektmitarbeiter enthalten nur noch die Attribute, welche die jeweilige Spezialisierung anzeigen (Std.-Satz und Projekterfahrung). Der Grund für diese Aufteilung war, die Vermeidung von zu vielen NULL-Werten in einer Gesamttafel. Um nun aber die Daten eines Projektmitarbeiters komplett anzeigen zu können, müssen die Tabellen *Mitarbeiter* und *Mitarbeiter_Projekt* wieder über einen JOIN miteinander verbunden werden.

```
Select * from Mitarbeiter
natural Join Mitarbeiter_Projekt
;
```

Um nun auf der externen Ebene (siehe Kap. 1.6) die Daten eines Projektmitarbeiters verarbeiten zu können, müsste immer dieser JOIN durchgeführt werden. Nun gibt es in SQL aber auch die Möglichkeit, definierte Abfragen zu einer sogenannten View (Sicht) zusammen zu fassen und als solche abzuspeichern. Nehmen wir das oben aufgeführte SQL-Statement und erstellen damit eine View mit neuem Namen.

```
Create View Projektmitarbeiter As
Select * from Mitarbeiter
natural Join Mitarbeiter_Projekt
;
```

Damit haben wir eine View mit dem Namen *Projektmitarbeiter* erstellt, welche wir nun wie eine „normale“ Tabelle verwenden können.

```
Select * from Projektmitarbeiter
;
```

Diese View wird aber nicht wie eine Tabelle in Postgres behandelt, sondern im jeweiligen Schema (Standard ist „public“) im Unterpunkt „Views“ angelegt. Schauen wir uns die neue View etwas genauer an, so sehen wir, dass hier das SELECT komplett ausgerollt hinterlegt wird.

```
-- View: public.projektmitarbeiter
```

```
-- DROP VIEW public.projektmitarbeiter;
```

```
CREATE OR REPLACE VIEW public.projektmitarbeiter AS
  SELECT mitarbeiter.pers_nr,
         mitarbeiter.vorname,
```

...
Dies bedeutet, dass wenn die View *Projektmitarbeiter* aufgerufen wird, wird das SELECT ausgeführt. Schauen wir noch genauer hin sehen wir, dass zu der View auch eine Regel (Rules) angelegt wurde. Diese Regel sieht folgendermaßen aus:

```
CREATE OR REPLACE RULE "_RETURN" AS
  ON SELECT TO public.projektmitarbeiter
    DO INSTEAD
      SELECT mitarbeiter.pers_nr,
```

...
Dies Regel besagt: Wird ein SELECT auf die View durchgeführt, so wird an deren Stelle das nachfolgende SELECT ausgeführt.

Diese bisher beschriebenen Sichten sind „reine Lesesichten“ (read only views). Dies bedeutet, dass damit nur Daten gelesen, aber nicht verändert werden können. Views, mit denen auch Daten eingefügt werden können, werden wir uns noch später anschauen.

Views können mit den bekannten SQL-Befehlen ALTER und DROP auch verändert oder gelöscht werden.

7.2.2 Datenunabhängigkeit durch Views

Betrachten wir nun einen Mitarbeiter in unserem Softwarehaus, der für das Kursmanagement der Mitarbeiter des Softwarehauses zuständig ist. Dieser handelt mit den externen Trainingsinstituten die Bedingungen aus und bucht auch die Kurse für die Mitarbeiter. Dieser Herr Klein braucht, um seine Aufgaben durchführen zu können, nun verschiedene Informationen aus der Datenbank. Wir generieren ihm dazu folgende Views:

1. Welcher Mitarbeiter hat welche Kurse besucht.
2. Welche Kurse bietet welches Trainingsinstitut an.
3. Welcher Mitarbeiter hat in diesem Jahr noch keinen Kurs besucht.

Exemplarisch wollen wir kurz die erste View anlegen.

```
Create View Mitarbeiter_besucht_kurs As
Select pers_nr, vorname, nachname, kurs_bez, termin_kurs, institut
from Mitarbeiter natural Join besucht_kurs
natural Join Kurs
;
```

Die anderen beiden Views zu erzeugen sollte Aufgabe der Studierenden sein.

Die Views sind erstellt und Herr Klein arbeitet erfolgreich damit. Jetzt wird es aber auf Grund organisatorischer Änderungen erforderlich, dass der konzeptionelle Datenbankentwurf des Softwarehauses erweitert werden muss. Zusätzlich zum Namen des Instituts sollen noch andere Merkmale in der Datenbank abgelegt werden und dazu wird nun eine neue Tabelle *Institut* angelegt. Diese erhält die *Inst-Nr* als künstlichen Primärschlüssel. Dieser wird dann als Fremdschlüssel in die Tabelle Kurs anstatt der Spalte *Institut* eingefügt. Dem Leser fällt natürlich sofort auf, dass nun alle Sichten für Herrn Klein umgeschrieben werden müssen. Die Programme aber, welche Herr Klein verwendet, in denen die Views aufgerufen werden, sollen aber nicht verändert werden. Dazu ist es erforderlich, dass die View denselben Namen behält und dieselben Attribute wie bisher liefert.

```
Create View Mitarbeiter_besucht_kurs As
Select pers_nr, vorname, nachname, kurs_bez, termin_kurs, institut
from Mitarbeiter natural Join besucht_kurs
natural Join Kurs
natural Join Training_Institut
;
```

Wir haben also gezeigt, dass sich der konzeptionelle Datenbankentwurf ändern lässt, ohne dass dadurch die externen Sichten geändert werden müssen. Diese Datenunabhängigkeit ist dann gewährleistet, wenn die Programme nicht auf Tabellen sondern auf Views zugreifen.

Diese Datenunabhängigkeit hat aber auch ihre Grenzen. Wird die Datentypdefinition von Attributen geändert oder werden Spalten gelöscht, so müssen meist auch die externen Programme geändert werden. Auch wenn Views so verändert werden, dass aus einer datenändernden Sicht eine reine Lesesicht wird (z.B. durch Hinzunahme von neuen Tabellen), dann müssen zumindest die Programme angepasst werden, über die bisher Daten eingefügt oder verändert wurden.

7.2.3 Über Sichten Daten verändern

Wenn wir für Herrn Klein eine View generieren, welche die Tabelle Kurs (vor der Erweiterung) genau abbildet, so kann über diese Sicht auch Daten in die Tabelle eingefügt werden.

```
CREATE VIEW V_kurs AS
```

```
SELECT kurs_nr, kurs_bez, institut
FROM kurs;
```

Der nachfolgende `INSERT` fügt Daten in die Tabelle Kurs über die View ein. Wir haben also hier eine View mit der wir auch Daten schreiben oder verändern können, da diese View die Tabelle mit allen Attributen 1:1 abbildet.

```
Insert into V_kurs
Values (2223, 'Testkurs', 'Testinstitut')
```

Nun haben wir in Kap. 7.2.2 das ERM erweitert und eine Tabelle *Institut* eingefügt. Um nun über die View *V_Kurs* die Datenunabhängigkeit zu gewährleisten, müssen wir diese wie nachfolgend beschrieben anpassen. Damit ist weiterhin ein `SELECT...` auf die Kursdaten möglich.

```
CREATE VIEW V_kurs AS
SELECT kurs_nr, kurs_bez, institut
FROM kurs JOIN training_institut USING (inst_nr)
;
```

Wollen wir aber nun wieder Kursdaten (mit dem obigen `INSERT`-Statement) einfügen, so erhalten wir eine Fehlermeldung, was natürlich auch einleuchtet. Will das DBMS die Daten einfügen ist es nicht mehr eindeutig, in welche Tabelle dies erfolgen soll und zusätzlich fehlen in der Tabelle *Training_Institut* auch weitere Werte für die definierten Not NULL-Attribute.

Dies bedeutet, dass eine Datenunabhängigkeit von Sichten nur bis zu einem gewissen Grad gewährleistet werden kann. Werden Views aus mehreren Tabellen gebildet, so wird aus der View eine „reine Lesesicht“.

Wir wollen uns diese Stelle noch kurz ansehen, welche Einschränkung eine View `read only` macht. Wie wir bereits gelernt haben, ist bei der Verbindung (`JOIN`) zweier Tabellen in der View, eine Veränderung der Daten nicht mehr möglich. Unter bestimmten Umständen kann dies jedoch umgangen werden und dazu liefert uns bereits die Fehlermeldung von Postgres beim `INSERT` einen Hinweis: ... , richten Sie einen *INSTEAD OF INSERT* Trigger oder eine *ON INSERT DO INSTEAD Regel ohne Bedingung*

Durch diesen Trigger wäre es möglich, anstatt der View andere Anweisungen auszuführen (verschiedene Inserts in Tabellen) um dadurch die Einschränkung der View zu umgehen.

Klar ist auch, dass wenn in der View eine Aggregation (`SUM`, `COUNT..`) stattfindet, dann keine Datenveränderungen über die View mehr stattfinden können.

Auch bei `DISTINCT`, `UNION` und Unterabfragen ist eine Datenveränderung über eine Sicht nicht mehr möglich.

Damit wollen wir das Kapitel der Data Retrieval Language (DRL) verlassen, von dem wir nur einen kleinen Ausschnitt behandeln konnten. Für eine praxisnahe tiefergehende Beschäftigung mit diesem Thema sei das Buch Ralf Adams [**Adam16**] empfohlen.

8 Das Transaktionskonzept (TCL)

8.1 Was versteht man unter einer Transaktion

Um die Problematik einer Transaktion zu beschreiben ist unser Softwarehaus leider nicht sehr geeignet. Hierzu müssen wir das ERM um einen Onlineshop oder dergleichen erweitern. Ein beliebtes Beispiel, welches immer wieder in der gängigen Literatur auftaucht, ist die Beschreibung einer Transaktion anhand einer Überweisung von einem Bankkonto auf ein anderes. Bisher haben wir Anweisungen erstellt, welche sich meistens völlig isoliert ausführen ließen. Das Einfügen eines neuen Kunden oder eines Mitarbeiters in unsere Datenbank waren Anweisungen welche kaum oder keine Abhängigkeiten zu anderen Anweisungen hatten. In vielen Anwendungen jedoch ist es erforderlich, dass verschiedene Anweisungen zusammen ausgeführt werden oder falls dies nicht möglich ist, dann darf keine der Anweisungen ausgeführt werden. Solch eine Abhängigkeit (Zusammengehörigkeit) von Anweisungen nennt man eine Transaktion.

Definition:

Eine Transaktion ist eine logische Arbeitseinheit (Bündelung mehrerer Anweisungen), welche dafür sorgt, dass logisch zusammengehörige Folgen von Operationen ohne negative Begleiterscheinungen auf der Datenbank ausgeführt werden können.

Dies bedeutet, dass eine Folge von Datenbankanweisungen zusammen eine Einheit bildet. Diese Anweisungen werden entweder alle, also vollständig ausgeführt, oder gar nicht. Wenn also im Verlauf einer Transaktion ein Fehler auftritt (z.B. eine Anweisung kann nicht ausgeführt werden), so werden auch die bisher erfolgten Anweisungen vollständig rückgängig gemacht. Der Zustand der Datenbank entspricht dann wieder dem Zustand vor dem Beginn der Transaktion.

Prinzipiell gibt es zwei unterschiedliche Anforderungen an ein DBMS im Zusammenhang mit dem Begriff der Transaktion.

1. Tritt während des Betriebs der Datenbank ein Fehler auf, sodass Datenbankoperationen nicht vollständig abgeschlossen werden konnten, so muss gewährleistet sein, dass die Datenbank in einem konsistenten Zustand bleibt (Recovery).
2. Greifen mehrere Benutzer gleichzeitig auf die Datenbank und eventuell sogar auf dieselben Datensätze zu, so muss auch hier gewährleistet sein, dass diese verschiedenen Transaktionen synchronisiert ablaufen und sich nicht gegenseitig behindern.

8.2 Verwaltung einer Transaktion

8.2.1 Befehle zur Steuerung einer Transaktion

Wird nichts anderes festgelegt, befindet sich das DBMS standardmäßig im AUTO-COMMIT Modus. Dieser kann mit `\SET AUTOCOMMIT = OFF` abgeschaltet werden.

Anweisung zum Einleiten einer Transaktion (BOT):

Die Anweisung `BEGIN { TRANSACTION }` (Postgres) kennzeichnet den Beginn einer Transaktion. Der AUTO-COMMIT Modus wird dadurch ausgeschaltet.

Ende der Transaktion:

Eine erfolgreiche Transaktion wird mit dem Befehl COMMIT abgeschlossen. Es kann auch END TRANSACTION angegeben werden. Alle Änderungen werden damit dauerhaft (persistent) in der DB eingespeichert.

Abbruch der Transaktion:

Eine Transaktion wird mit der Anweisung ROLLBACK aufgegeben und die Datenbank in den ursprünglichen Zustand (vor Beginn der Transaktion) zurückgesetzt. Nach einem COMMIT oder ROLLBACK beginnt eine neue Transaktion. Das DBMS befindet sich wieder im AUTO-COMMIT Modus.

8.2.2 ACID

Wie bereits erwähnt, überführt eine Transaktion eine Datenbank von einem konsistenten Zustand in einen neuen konsistenten Zustand.

Dazu muss die Transaktion ganz bestimmten Anforderungen genügen, welche unter dem Kürzel ACID zusammengefasst werden.

➤ **Atomicity** (Atomarität)

Damit ist gemeint, dass eine Transaktion die kleinste, nicht mehr sinnvoll weiter unterteilbare Einheit darstellt. Eine Transaktion erscheint von außen als eine Einheit.

➤ **Consistency** (Konsistenz)

Die Datenbank befindet sich immer in einem konsistenten Zustand. Falls dieser Zustand nach Ausführung verschiedener SQL-Anweisungen nicht erreicht werden kann, wird die DB wieder in den vorherigen Zustand zurückgesetzt (rollback).

Wenn Änderungen am Ende der Transaktion die logische Konsistenz (wird anhand von Integritätsbedingungen etc. geprüft) verletzen, wird die Transaktion zurückgewiesen. „Während“ der Transaktion kann die Konsistenz verletzt werden.

➤ **Isolation**

Verschiedene Transaktionen laufen voneinander isoliert ab und beeinflussen sich dadurch nicht gegenseitig. Die Ergebnisse einer Transaktion sind bis zu ihrem Ende (commit) für andere Transaktionen unsichtbar. Den Grad der Unsichtbarkeit wird durch die sogenannte „Isolationsebene“ bestimmt.

➤ **Durability** (Dauerhaftigkeit)

Alle Aktionen einer abgeschlossenen Transaktion (nach einem commit) bleiben dauerhaft (persistent) in der DB erhalten. Auch ein Systemabsturz kann dies nicht mehr gefährden.

8.3 Das Protokoll einer Transaktion

Wie werden nun die Anforderungen der Sicherstellung konsistenter Daten vom DBMS gewährleistet? Zuerst einmal müssen Änderungen auf der Datenbank mitprotokolliert werden.

Dazu gibt es eine Log-Datei (in ihr erfolgt die Protokollierung) durch die sichergestellt wird, dass in einem Fehlerfall keine Änderungen und sonstige Aktionen auf der Datenbank verloren gehen. Dies ist unabhängig von der Art des Fehlers. Egal ob es sich um einen SQL-Fehler, einen Stromausfall oder einen Servercrash handelt.

Zuerst wird jede Änderung (Insert, Update, Delete) in einen Hauptspeicherpuffer geschrieben und anschließend zeitlich versetzt in eine Datei ausgelagert. Hier werden aber nicht SQL-Statements protokolliert, sondern Änderungen auf der Datensatzebene. Dieser Log-Puffer (oder Datei) enthält alle Informationen die erforderlich sind um die Änderungen im Falle eines Fehlers rückgängig zu machen. Weiter in die Tiefe wollen wir zu diesem Thema nicht gehen, denn diese Informationen sollten ausreichen um das Prinzip grob zu verstehen.

Dafür wollen wir uns nun verschiedene Beispiele ansehen, welche die oben beschriebenen grundsätzlichen Anforderungen an eine Transaktion verdeutlichen.

8.4 Der Online-Shop

8.4.1 Erweiterung des ERM

Wie oben bereits angedeutet, erweitern wir nun das ERM des Softwarehauses um einen Online-Shop. In diesem Shop können sich Kunden verschiedene Artikel auswählen, dies in einer Bestellung zusammenfassen und anschließend dann kaufen. Ein Artikel liegt physisch an einem bestimmten Lagerort.

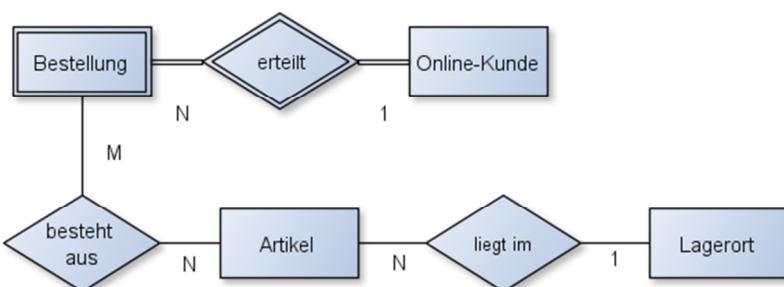


Abbildung 41: Bestellwesen im Softwarehaus

Relationen aus dem ERM:

Online-Kunde: $\{ \underline{\text{Kunde ID}}: \text{serial}, \text{Nachname: string}, \text{Vorname: string}, \dots \}$

Bestellung: $\{[Bestellung_ID: \text{serial}, \text{Kunde_ID: integer}, \dots, \text{Status: string}]\}$

Artikel: {[*Artikel_ID*:serial, *Bezeichnung*: string, ...]}

Lagerort: {[Artikel_ID:integer, *Lagerort*: string, *Menge_mindest*:decimal,
Menge_aktuell:decimal]}

Bestellung_position: {[Position_nr:integer, Bestellung_ID: integer, Artikel_ID: integer, Menge:decimal]}}

8.4.2 Warenbestellungen

Beispiel 1:

Will nun ein Kunde Ware im Online-Shop bestellen, so müssen mehrere Datensätze in unsere Datenbank geändert werden. Dies sind dann im Einzelnen:

1. Ein neuer Datensatz wird in die Tabelle *Bestellung* eingefügt.
 2. Für jeden Artikel wird ein Datensatz in die Tabelle *Bestellung_position* (Relation der Beziehung *besteht_aus*) eingefügen.
 3. Dazu muss zuerst die neue *bestellung_id* aus der Tabelle Bestellung ermittelt werden.
 4. Die *positions_nr* wird in der Tabelle *Bestellung_position* beim Einfügen hochgezählt (in unserem Beispiel nicht automatisch).

Um dies auf der Datenbank durchzuführen sind nachfolgende SQL-Statements erforderlich.

```
-- Bestellung anlegen
INSERT INTO bestellung (kunde_id, adresse)
VALUES (1, 'Am Wald 3')
;
-- Artikel in Tabelle Bestellung_position einfügen
INSERT INTO bestellung_position (position_nr, bestellung_id,
artikel_id, menge)
VALUES
(1, CURRVAL('bestellung_bestellung_id_seq'), 2, 1),
(2, CURRVAL('bestellung_bestellung_id_seq'), 3, 1);
```

Hierbei wird mit dem Befehl CURRVAL ('bestellung_bestellung_id_seq') der aktuelle Wert des Attributs von *Bestellung_Id* (es handelt sich um ein Autoincrement-Attribut) ausgelesen und an die Tabelle (*Bestellung_position*) übergeben.

Wenn nun, aus welchen Gründen auch immer, das erste Statement nicht ausgeführt wird (z.B. wir haben den Tabellennamen falsch geschrieben „bestelung“), so verhindert das DBMS, dass auch die zweite Anweisung ausgeführt wird. Anstatt eine inkonsistente Datenbank zu hinterlassen, wird eine Fehlermeldung ausgegeben. Das DBMS regelt dieses Problem aufgrund der hinterlegten Integritätsregeln.

Beispiel 2:

Nehmen wir nun an, die Bestellung war erfolgreich und beide Artikel wurden bestellt. Als nächstes müssen nun beide Artikel versendet werden.

In der Tabelle *Bestellung* muss im Feld *Status* der Wert auf „versendet“ gesetzt werden. Des Weiteren muss der Lagerbestand (*menge_aktuell*) in der Tabelle *Lager* um die jeweilige Bestellmenge verringert werden.

```
UPDATE Lager
SET menge_aktuell = menge_aktuell - 1
WHERE artikel_id = 2
;
UPDATE bestellung SET status = 'versendet'
WHERE bestellung_id = 1;
```

Nehmen wir nun wieder an, dass eine der beiden Anweisungen aus irgendeinem Grund nicht ausgeführt wird. Was wird nun passieren?

Entweder:

Der Lagerbestand wird verringert, aber die Bestellung hat immer noch den Status „offen“. Wird also vermutlich nochmals versendet.

oder:

Die Artikel werden aus dem Lagerbestand entnommen, aber der Lagerbestand wird nicht verringert.

In jedem Fall befindet sich die Datenbank in einem inkonsistenten Zustand und dies gilt es zu verhindern.

Forderung: Beide Anweisungen müssen in jedem Fall gesamt ausgeführt werden. Kann eine der Anweisungen nicht ausgeführt werden, darf die andere auch nicht ausgeführt werden.
 Wie wir bereits wissen, nennt man diese Bündelung von Anweisungen eine **Transaktion**.

8.4.3 Ablauf einer Transaktion

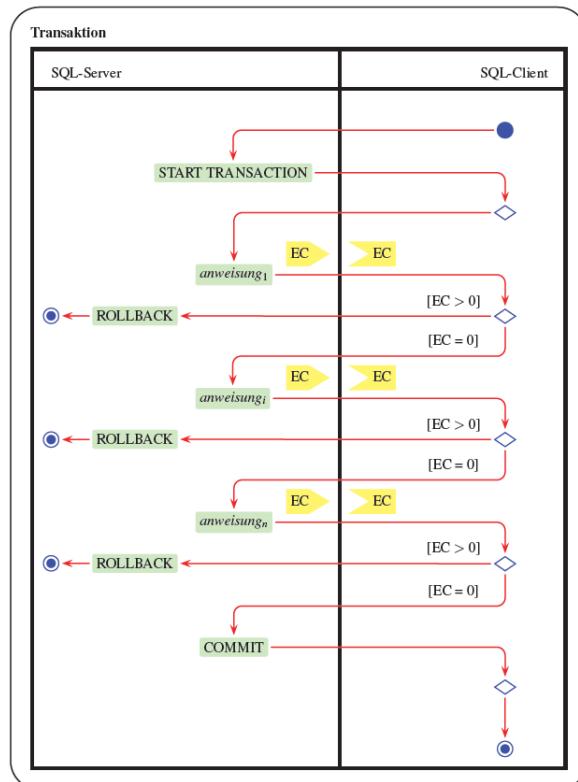


Abbildung 42: Schematischer Ablauf einer Transaktion

(Quelle: [Adam16, S.292])

Wie eine solche Transaktion auf Client und Server abläuft, veranschaulicht schematisch die **Abbildung 42**.

1. Die Transaktion wird vom Client durch `Start Transaction` initialisiert.
2. Auf dem Server wird nun die Umgebung so eingerichtet, dass bei einem Abbruch der Transaktion auf den ursprünglichen Zustand zurückgesetzt werden kann (Log...). Die Steuerung wird an den Client zurückgegeben.
3. Vom Client wird nun eine SQL-Anweisung an den Server gesendet und dieser führt diese aus. Er sendet, abhängig von der erfolgreichen Ausführung der Anweisung einen Error Code an den Client zurück.
4. Dieser entscheidet anhand des Codes ob die nächste Anweisung ausgeführt werden kann, oder mit einem `Rollback` die Transaktion abgebrochen wird.
 Bei einem Rollback stellt der Server den ursprünglichen Zustand wieder her.
5. Wurden alle Anweisungen ohne Fehler ausgeführt, sendet der Client eine `Commit` an den Server und dieser schließt die Transaktion ab. Dies bedeutet, dass nun alle Änderungen in die Datenbank geschrieben werden.

Beispiel einer Transaktion (mit Begin und commit)

:

Begin;

UPDATE Lager

SET menge_aktuell = menge_aktuell - 1

WHERE artikel_id = 2

;

UPDATE bestellung SET status = 'versendet'

WHERE bestellung_id = 1

;

Commit;

8.5 Konkurrierende Zugriffe

8.5.1 Voraussetzung

Wie in Kap. 8.1 bereits angedeutet, gibt es noch eine zweite Anforderung im Zusammenhang mit dem Begriff der Transaktion. Dies ist die Synchronisation von konkurrierenden Zugriffen mehrerer Clients auf den Datenbestand. Dies verdeutlicht die nachfolgende Abbildung.

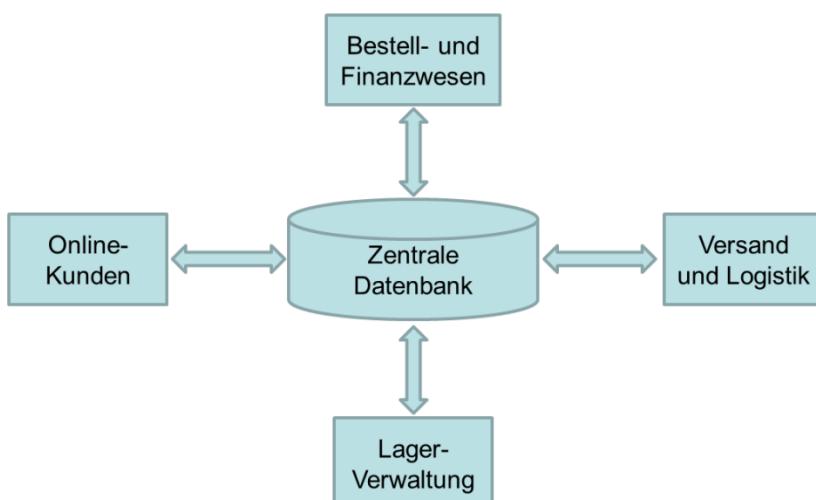


Abbildung 43: Konkurrierende Zugriff auf eine Datenbank

Vergegenwärtigen wir uns einmal die Situation innerhalb einer Firma bzgl. einer zentralen Datenhaltung. Alle Bereiche der Firma, wie oben dargestellt, halten ihre Daten in einer Datenbank. Dies hat den Grund, da die Daten nicht isoliert existieren, sondern für alle transparent sein müssen. Überlegen wir uns nun ein Szenario welches bei dieser Konstellation auftreten kann.

1. Ein Kunde nennen wir ihn Nr.1, interessiert sich für ein Office-Produkt (Artikel Nr.3) und nimmt dieses in seine Bestellung auf.
Um zu gewährleisten, dass dieser Artikel noch am Lager ist und somit sofort lieferbar ist, wird die Menge des Lagerbestands abgefragt (es ist noch ein Artikel vorhanden). Es wird angezeigt, dass der Artikel lieferbar ist.

2. Zur selben Zeit interessiert sich ein weiterer Kunde u.a. für den Artikel mit der Nr. 3 und will wissen, ob dieser sofort lieferbar ist. Auch ihm wird angezeigt, dass noch ein Artikel im Lagerbestand vorhanden ist.
3. Nehmen wir nun an, Kunde 2 schließt seine Bestellung vor Kunde 1 ab, so erhält dieser den Artikel und damit ist kein Artikel mit der Nummer 3 mehr im Lager. Dieser Artikel ist vom Großhändler derzeit auch nicht lieferbar.
4. Dem Kunden (Nr.1) wurde aber mitgeteilt, dass der Artikel sofort lieferbar ist. Seine Bestellung beruht daher auf nicht mehr aktuellen Daten.

Wir sprechen hier von konkurrierenden Zugriffen auf denselben Datenbestand. Dies ist ein ganz normaler Vorgang in einem DBMS. Damit ein Problem wie oben beschrieben nicht vorkommen kann, müssen wir den Ablauf in eine Transaktion einbinden. Nun kann es bei sehr vielen gleichzeitigen Bestellungen vorkommen, dass mehrere Transaktionen aufgeführt werden.

Wenn Transaktionen parallel und gleichzeitig ausgeführt werden, spricht man auch von Nebenläufigkeit.

Bei ACID gab es die Forderung nach Isolation, also dass sich die Transaktionen nicht gegenseitig beeinflussen.

Nachfolgend sollen nun für parallel laufende Transaktionen (konkurrierenden Zugriff) verschiedene Szenarien dargestellt werden. In diesen Szenarien arbeiten wir mit gleichzeitiger Verringerung und Erhöhung des Lagerbestandes.

8.5.2 Beispiele konkurrierender Zugriffe

1. Szenario:

- Die Abteilung Versand entnimmt einen Artikel (Nr. 3) und verringert den Lagerbestand.
- Die Lagerverwaltung arbeitet parallel und verbucht gleichzeitig neue Ware zum Artikel (Nr. 3).

Versand: (T1)

1. Lesen `menge_aktuell = 5`
3. `menge_aktuell = menge_aktuell - 1`
5. Schreiben `menge_aktuell = 4`

Lager: (T2)

2. Lesen `menge_aktuell = 5`
4. `menge_aktuell = menge_aktuell + 10`
6. Schreiben `menge_aktuell = 15`

- Der Lagerbestand ist falsch, da die erste Entnahme verloren ging.
- Es entstand der Verlust einer Modifikation oder engl. lost update.

2. Szenario:

Versand: (T1)

1. Lesen `menge_aktuell = 5`
2. `menge_aktuell = menge_aktuell - 1`
3. Schreiben `menge_aktuell = 4`
7. **ROLLBACK**
8. `menge_aktuell = 5`

Lager: (T2)

4. Lesen `menge_aktuell = 4`
5. `menge_aktuell = menge_aktuell + 10`
6. Schreiben `menge_aktuell = 14`

- Der Lagerbestand der Abt. Lager beruht nach Rücksetzen der Änderung auf falschen Daten.
- Das Lesen „ungültigen“ Daten wird dirty read genannt

3a. Szenario:

Versand: (T1)

1. Lesen `menge_aktuell = 5`
3. ... (keine Update)
6. Lesen `menge_aktuell = 15`

Lager: (T2)

2. Lesen `menge_aktuell = 5`
4. `menge_aktuell = menge_aktuell + 10`
5. Schreiben `menge_aktuell = 15`

- Wiederholte Abfrage derselben Tabellenzeilen führt zu unterschiedlichen Ergebnissen.
- Dieses Lesen „ungültigen“ Daten wird non repeatable read genannt.

3b. Szenario:

Versand: (T1)

1. `count(*) from lager (4)`
2.
3. `count(*) from lager (5)`

Lager: (T2)

3. `insert into lager value(5,...)`

- Szenario 3b entspricht im Prinzip dem Szenario 3a, aber ein neuer Artikel taucht durch Transaktion 2 (Lager) auf.
- Dies wird **Phantom** (Phantomproblem) genannt, da aus dem „Nichts“ neue Zeilen auftauchen.

8.5.3 Isolationsstufen

Die oben geschilderten Probleme würden nicht entstehen, wenn alle Transaktionen seriell (nacheinander) ablaufen würden. Dies ist aber oft nicht möglich und in Hinsicht auf die Performance auch nicht wünschenswert. Deshalb muss das DBMS in der Lage sein die beschriebenen Probleme zu verhindern.

Dazu werden Zugriffe auf Daten von anderen Transaktionen (Programmen) temporär verhindert, in dem Datensätze gesperrt werden.

Hierfür gibt es die Möglichkeit verschiedene Sperrumfänge (Zeilen, Tabellen, Seiten) festzulegen. Über die sogenannten Isolationsstufen wird festgelegt, in welchem Maße Leseoperationen (SELECT) einer Transaktion gegen konkurrierende Änderungen geschützt werden.

Im SQL-Standard sind bei einer Transaktion vier Isolationsebenen festgelegt.

Davon ist SERIALIZABLE die am strengsten isolierte (alle Transaktionen laufen nacheinander ab). Dies ergibt, wie bereits erwähnt, aus pragmatischen Gründen wenig Sinn, daher muss in Regelfall der Grad der Nebenläufigkeit erhöhen werden.

Prinzipiell kann man sagen, dass der Isolationsgrad einer Transaktion angibt, welches der Phänomene (dirty read, etc....) das DBMS zulässt und welches er verhindern soll.

1. **Read uncommitted:** Ist in Postgres die Voreinstellung. Erlaubt ein dirty read .
2. **Read committed:** Ist in Postgres die Voreinstellung. Erlaubt das nicht wiederholbare Lesen (non repeatable read).
3. **Repeatable read:** Ermöglicht das Lesen von Phantomien.
4. **Serializable:** Erlaubt kein paralleles Lesen.

Kurz nochmals die einzelnen Stufen der konkurrierenden Zugriff zusammengefasst.

- **dirty read:** Lesen von Daten welche von einer gleichzeitigen, uncommitted Transaktion geschrieben wurden.
- **nonrepeatable read:** Das wiederholte Lesen von Daten durch eine Transaktion welche von einer anderen Transaktion zuvor verändert wurde (commit seit dem initialen Lesen).
- **phantom read:** Beim nochmaligen Ausführen einer Abfrage durch eine Transaktion, entspricht das Ergebnis der Zeilen nicht mehr dem zuvor ermittelten Wert, da eine andere Transaktion zuvor dies geändert (commit) hat (z.B. neue Zeile eingefügt).

Isolationsgrad	dirty read	non repeatable read	phantoms
Read uncommitted	ja	ja	ja
Read committed	nein	ja	ja
Repeatable read	nein	nein	ja
Serializable	nein	nein	nein

Abbildung 44: Isolationsgrad der Sperrumfänge

Die Isolationsstufe muss nach dem Beginn der Transaktion festgelegt werden. Dies erfolgt durch das Statement:

```
\SET TRANSACTION ISOLATION LEVEL <Isolationsgrad>
```

Für diejenigen unter den Studierenden welche sich etwas eingehender mit der Materie der konkurrierenden Zugriffe beschäftigen möchten, nachfolgend noch einige kurze Informationen, um sich dann selbstständig tiefer einzuarbeiten.

PostgreSQL stellt den Entwicklern verschiedene Werkzeuge zur Verfügung um konkurrierende Zugriffe auf die Daten zu verwalten. Intern wird die Datenkonsistenz durch die Verwendung des MVCC (Multiversion Concurrency Control) gewährleistet.

Die Grundidee hinter einer Multiversion Concurrency Control (MVCC) Architektur ist es, dass jeder Benutzer der Datenbank einen eigenen Snapshot der Datenbank zu einem bestimmten Zeitpunkt sieht. Dazu werden intern mehrere Versionen eines Objektes (z. B. eines Tupels) gehalten, welche durch fortlaufend erhöhte Transaktionsnummern voneinander unterschieden werden. Dies bedeutet, dass jede Transaktion auf einem ihr zur Verfügung gestellten Snapshot der Tabellen arbeitet. Dies verhindert es, dass innerhalb einer aktiven Transaktion inkonsistente Daten, welche von einer konkurrierenden Transaktion verändert wurden, sichtbar sind.

Mit MVCC wird die übliche Locking-Methode von Datenbanksystemen vermieden und damit auch die Sperreskalationen, wie beispielsweise Deadlock umgangen.

Natürlich existiert in PostgreSQL auch die Möglichkeit der Sperrung von Tabellen, Seiten und Zeilen um die Kontrolle einer vollständigen Isolation einer Transaktion zu erreichen und damit jeden Zugriffskonflikt zu vermeiden.

Diese Ausführungen sollen ausreichen um das Prinzip der Transaktion und Sperrproblematik grob zu verstehen.

9 Indizierung von Tabellen

In Kap. 5 und Kap. 7 haben wir uns damit beschäftigt wie wir mit Hilfe von SQL-Befehlen Daten in einer Datenbank speichern und wieder abrufen können. Wie diese Operationen vom DBMS durchführt und organisiert werden, hat uns zu diesem Zeitpunkt noch nicht interessiert. Nun wird es an der Zeit dem Datenbankmanagementsystem mal etwas unter die „Haube“ zu schauen. Zuerst aber wollen wir uns grundsätzlich mit den internen Speichermöglichkeiten von Daten beschäftigen.

9.1 Speicher- und Zugriffsmöglichkeit von Daten

Für die Speicherung und den Zugriff auf Daten entstanden historisch verschiedene Möglichkeiten, welche tlw. auch heute noch Verwendung finden:

9.1.1 Der sequentielle Zugriff

Dieses ist die einfachste Form der Datenspeicherung. Die Datensätze werden nach einem Auswahlkriterium sortiert und nacheinander auf dem Speichermedium abgelegt (z.B. Bandlaufwerk). Diese können unterschiedliche Längen haben, ihnen ist jedoch keine physikalische Adresse zugeordnet, so dass für die Suche nach einem bestimmten Satz (Record) stets der gesamte Datenbestand vom Anfang an gelesen werden muss, bis der richtige Datensatz gefunden ist.

Die Zugriffsanzahl beträgt hierbei im Mittel: Anzahl der Sätze (n)/2.

Ist jeder Satz in einem bestimmten Block abgelegt, so kann durch binäres Suchen die Anzahl der Zugriff auf $Z = \log_2 n$ Zugriffe reduziert werden.

Das Zeitverhalten wird bei dieser Zugriffsart dadurch negativ beeinflusst, dass beim Einfügen und Löschen eines Datensatzes, der gesamte Datenbestand ab Satz $i+1$ verschoben werden muss.

9.1.2 Der direkte Zugriff

Auch hier werden die Daten in Sätze (Records) unterteilt abgelegt, aber jedem wird eine Adresse zugeordnet. Unter dieser Adresse kann ein Satz direkt angesprochen, gelesen und gegebenenfalls geändert werden.

Dies ist bereits wesentlich komfortabler und wird auch heute tlw. noch angewendet.

9.1.3 Index-sequentieller Zugriff

Diese Speicherung der Daten wurde in den 60-iger Jahren von der Firma IBM entwickelt. Hierbei werden die Daten sequentiell in Blöcken abgespeichert. Innerhalb eines Blocks sind die Sätze nach ihren Schlüsseln sequenziell geordnet. Der Zugriff auf die Blöcke erfolgt über eine Indextabelle in der die Adresse eines Blocks (des größten Schlüsselwerts im Block) abgelegt ist.

Beim Zugriff auf einen bestimmten Datensatz wird zuerst in der Indextabelle nach dem Schlüssel gesucht und so die Adresse des Blocks gefunden, in dem nun binär oder sequentiell nach dem gewünschten Datensatz gesucht werden kann.

Diese Speicherstruktur erlaubt einen schnellen Zugriff auf die benötigte Information. Um aber beim Einfügen und Löschen von Datensätzen ein fortwährendes Verschieben der gespeicherten Sätze auszuschließen, werden neue Sätze in einem sogenannten Überlaufbereich abgelegt. Wird beim Suchen der gewünschte Datensatz im Block nicht gefunden, so muss die Suche im Überlaufbereich

fortgesetzt werden. Dies führt mit größer werden des Überlaufbereichs zu einer Erhöhung der Suchzeit, wodurch von Zeit zu Zeit eine Reorganisation des Datenbestands erforderlich wird. Dabei werden die im Überlaufbereich gespeicherten Sätze in die Blöcke einsortiert.

Diese Zugriffsmethode findet auch bei Datenbankverwaltungssystemen Verwendung.

Zur Organisation dieser Zugriffsmethode werden sogenannte Bayer-Bäume verwendet (vgl. [Wirt91], S.326).

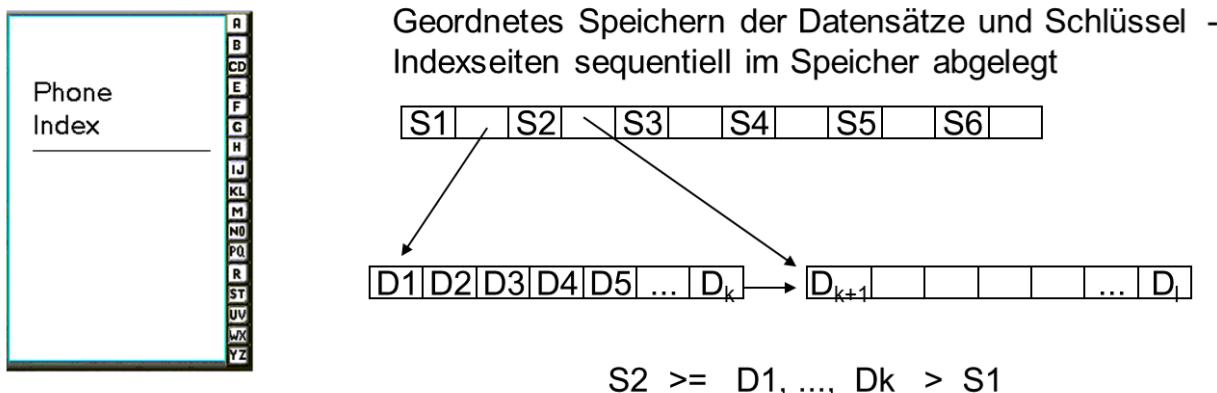


Abbildung 45: Index-sequentielle Speicherung
[siehe Folien Prof. Dr. Dirk Reichardt]

9.1.4 Baumstruktur

Ein Baum besteht aus einer Hierarchie von Datenelementen, welche Knoten genannt werden. Die oberste Ebene wird Wurzel-Knoten oder Wurzel (root) bezeichnet.

Jeder Knoten, mit Ausnahme der Wurzel, hat ein übergeordnetes Element welches Eltern-Knoten (parent) oder Vater genannt wird. Die Zuordnung eines Knoten darf nur zu einem Elternelement erfolgen, aber ein Element kann mehrere untergeordnete Elemente (Kinder, children) besitzen. Es besteht also eine 1:n- Beziehung zwischen Eltern und Kinder. Diese Abbildung von Daten wird auch als hierarchisch bezeichnet, wenn die Beziehung der Elemente untereinander eine Baumstruktur aufweist

9.1.5 Schlüsselbaum (B-Baum)

Die B-Bäume, oder auch Mehrwegbäume genannt, wurden 1970 von R. Bayer entwickelt und ähneln in ihrem Aufbau sehr stark den Binärbäumen. Der einzige Unterschied besteht darin, dass hier mehrere Schlüsselbegriffe in einem Knoten zusammengefasst sind. Es verringert sich dadurch die Anzahl der Zugriffe auf die in den Blöcken gespeicherten Datensätze.

Alle Schlüsseleinträge in einem Knoten sind aufsteigend sortiert. Hat ein Knoten keine Kinder so wird er Blatt-Knoten genannt. In jedem Knoten sind sowohl Zeiger auf dessen Kinder als auch weiter Informationen gespeichert (siehe **Abbildung 46**). B-Bäume sind immer ausgeglichen (balanced). Bei einem Blattknoten sind die Zeiger auf die Kindknoten alle NULL.

Ein Baum mit der Ordnung n (Mindestanzahl der Schlüssel pro Seite) und der Höhe h (Anzahl Stufen ohne Wurzel) wird als Baum des Typs (n,h) bezeichnet. Der Baum in **Abbildung 47** ist also ein Baum des Typs (1,2).

Man unterscheidet in der Literatur zwischen B-, B⁺- und B^{*}-Bäumen. B- und B^{*}-Bäume unterscheiden sich nur dadurch wann ein Knoten geteilt wird. Dies ist bei B-Bäumen bei $>n/2$ und bei B^{*}-Bäumen bei $>2/3 n$. Manchmal findet man auch B^{*}- und B⁺-Bäume synonym genannt, was aber je-

doch nicht richtig ist. B^+ -Bäume haben in den Knoten keine Zeiger auf die eigentlichen Daten. Diese sind ausschließlich in den Blättern gespeichert, welche man auch Index-Knoten nennt. B^+ -Bäumen dienen zur Verwaltung der Index-Strukturen in Datenbanken. Wir sprechen aber nachfolgend einfach vom B-Baum ohne diese Unterscheidung zu berücksichtigen.

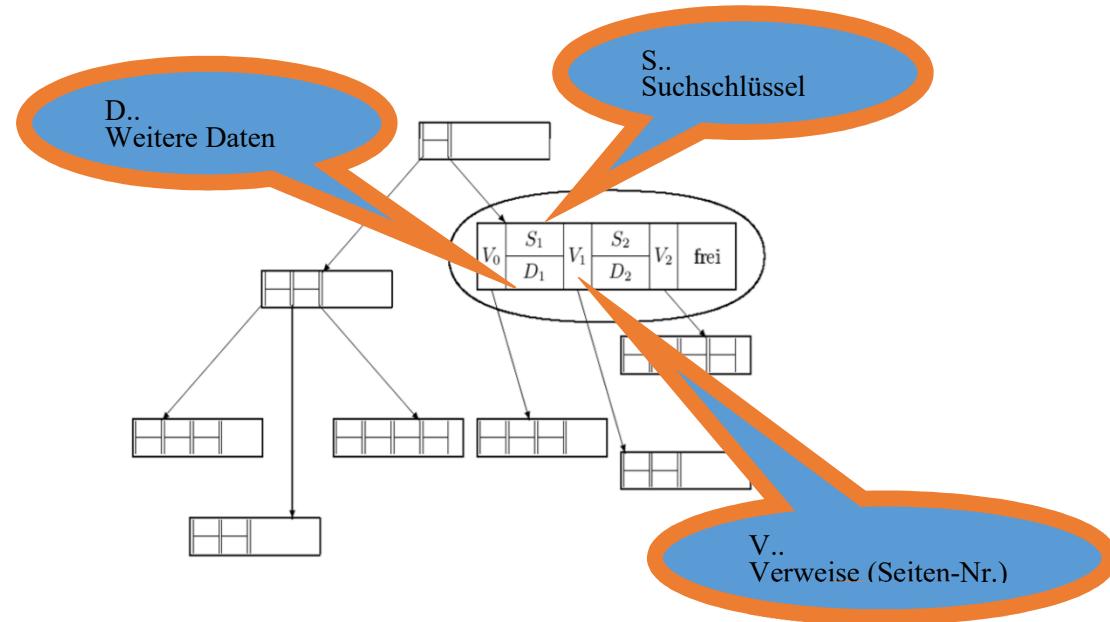


Abbildung 46: Schlüssel und Zeiger im Knoten

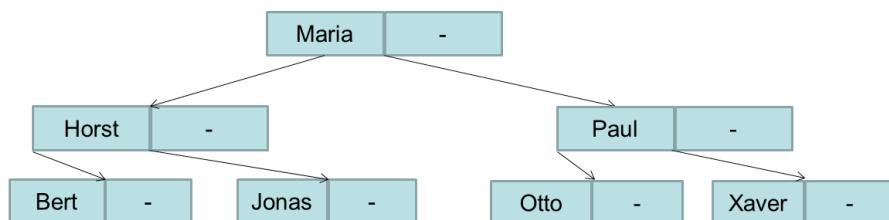


Abbildung 47: B-Baum

9.2 Indizes

9.2.1 Index auf Primärschlüssel

Der Zugriff auf Datensätze in der Datenbank erfolgt also über einen Index (B-Baum). Dies ist aber nur die halbe Wahrheit, denn nur wenn wir über den festgelegten Primärschlüssel auf einen Datensatz zugreifen, wird der automatisch angelegt Index (bei `Create Table`) verwendet. Der Zugriff auf alle anderen Attribute der Tabelle erfolgt über einen Tabellenscan. Dies bedeutet, dass die Tabelle sequentiell nach dem gewünschten Attribut durchsucht wird. Dies kann bei sehr vielen Datensätzen sehr zeitaufwändig sein.

Der Index hat den Vorteil, dass über wenige Zugriffe (im B-Baum) der gewünschte Datensatz gefunden werden kann. Dies hat dann natürlich Auswirkung auf die Performance beim Zugriff auf Datensätze.

9.2.2 Verwendung weiterer Indizes

Wie gerade besprochen, kann der Zugriff auf ein beliebiges Attribut (mit Ausnahme des Primärschlüssels) sehr performanceintensiv sein. Stellen wir nun bei der Verwendung der Tabellen des Softwarehauses fest, dass bei der Suche sehr häufig auf ein bestimmtes Attribut zugegriffen wird, so muss überlegt werden, ob auf dieses Attribut (Tabellenspalte) nicht ein zusätzlicher Index gelegt wird.

Hier ist allerdings folgendes zu beachten:

- Wird ein neuer Datensatz in eine Tabelle eingefügt, so müssen bei diesem zusätzlichen Index auch die zugehörigen Werte im B-Baum eingefügt werden. Dies kann in bestimmten Fällen zu einer Reorganisation des gesamten Index führen (z.B. Knoten sind voll und müssen geteilt werden) was zusätzlich Performance beansprucht.
- Alle zusätzlich angelegten Indizes einer Tabelle werden also bei allen Änderungsoperationen automatisch aktualisiert. Dies kann bei vielen (eventuell vorschnell angelegten) Indizes zu einem Zusatzaufwand bei der Indexaktualisierung führen. Ein Index sollte daher nicht leichtfertig angelegt werden.

9.2.3 Erstellen eines Index

Nun haben wir genug theoretisch über Indizes gesprochen und wir wollen nun einen zusätzlichen Index anlegen. Der SQL-Befehl dazu lautet:

```
CREATE INDEX [Index_Name]
ON TabellenName
(Attribut);
```

Der `Index_Name` muss nicht zwingend angegeben werden. Falls keiner angegeben wurde, wird vom DBMS selbstständig ein Name vergeben. Das `Attribut` bezeichnet den Spaltennamen für den ein Index angelegt werden wird. Hier können auch mehrere Spalten einer Tabelle angegeben werden, welche zusammen den Index bilden.

An welcher Stelle in der Datenbank des Softwarehauses gibt es nun am meisten Sinn einen zusätzlichen Index in einer Tabelle zu vergeben? Wenn wir uns überlegen, dass vielleicht von der Buchhaltung immer wieder aus abrechnungstechnischen Gründen abgefragt wird, welche Projektmitarbeiter haben zu bestimmten Projekten welche Leistungen erbracht, dann lautet die Abfrage hierzu z.B. folgendermaßen:

```
Select Leistung_Nr, Auftrag_nr from leistung
where zu_projekt = 3050
;
```

Ausgehend von dieser Abfrage müssen jetzt noch die zugehörigen Projektmitarbeiter gesucht werden, welche auf diese Leistung gebucht haben. Somit wäre es dann möglich die gesamten Leistungen zu einem Projekt aufzuaddieren. Wir wollen uns nun einmal ansehen, wie unser DBMS und hier die Komponente des sogenannten Optimizer (Abfrageoptimierer) diese Abfrage ausführt. Dazu stellen wir dem Select-Statement ein Explain vorweg, wodurch uns der Optimizer anzeigt, wie der Befehl ausgeführt wurde.

```
explain select ...
;
```

	Data Output	Explain	Notifications
	QUERY PLAN		
	text		
1	Seq Scan on leistung (cost=0.00..1.19 rows=6 width=8)		
2	Filter: (zu_projekt = 3050)		

Wie erwartet wird die Tabelle sequentiell (Tabellenscan) nach dem gesuchten Projekt durchsucht, da nur auf den Primärschlüssel ein Index existiert.

Angenommen diese Zugriffe würden sehr häufig erfolgen und die Anzahl der Projekte wären auch entsprechend groß, so gäbe es durchaus Sinn, auf das Attribut `zu_projekt` (Projektnummer) einen Index zu legen. Dieses wollen wir mit dem nächsten Befehl auch tun.

```
Create index PR_Nr_Leistung
on leistung
(zu_projekt)
;
```

Wir können uns diesen neuen Index nun auch anzeigen lassen. Dazu bietet PostgreSQL aber keinen Befehl wie z.B. `show Indexes` in MySQL, sondern hier muss die Tabelle *PG-Indexes* abgefragt werden.

```
select * from PG_Indexes
where tablename ='leistung'
;
```

[Data Output](#) [Explain](#) [Notifications](#)

	schemaname name	tablename name	indexname name	tablespace name	indexdef text
1	public	leistung	leistung_pkey	[null]	CREATE UNIQUE INDEX leistu...
2	public	leistung	pr_nr_leistung	[null]	CREATE INDEX pr_nr_leistung ...

Hier sehen wir nun neben dem Primärindex den von uns neu angelegten Index *pr_nr_leistung*. Nun wollen wir im nächsten Schritt diesen Index auch gleich verwenden und suchen wieder unser Projekt „3050“. Hier erleben wir nun eine kleine Überraschung, denn der Optimizer verwendet leider den angelegten Index nicht und führt wieder einen Tabellenscan durch.

Wie ist dies nun zu erklären? Der Optimizer sucht sich den für ihn günstigsten (Cost) Weg aus, um die Daten zusammenzustellen und dies ist bei unserer Tabelle nicht die Verwendung des neuen Index, sondern weiterhin die Tabelle sequentiell zu durchsuchen.

Grundsätzlich gilt: Ob und wenn ja, in welcher Form ein Index zum Datenbankzugriff verwendet wird, ist die Entscheidung des DBMS und damit des Optimizers. Wir können zwar Indizes anlegen, aber die Verwendung liegt nicht in unserer Hand.

Im nächsten Schritt wollen wir aber doch noch versuchen die Verwendung eines Index durch den Optimizer zu erreichen. Wie bereits erwähnt, gibt die Anwendung eines Index bei großen Tabellen durchaus einen Sinn und damit einen Performancevorteil.

Um dies zu testen verwenden wir die im Internet frei verfügbare Datenbank „World_DB“ in der mehrere tausend Städte und die dazugehörigen Länder eingetragen sind.

Nachfolgend ein kleiner Ausschnitt der Tabelle „City“ der Datenbank.

[Data Output](#) [Explain](#) [Notifications](#)

	id [PK] integer	name text	countrycode character (3)	district text	population integer
1	1	Kabul	AFG	Kabol	1780000
2	2	Qanda...	AFG	Qandahar	237500
3	3	Herat	AFG	Herat	186800
4	4	Mazar...	AFG	Balkh	127800
5	5	Amster...	NLD	Noord-Holl...	731200

Wir suchen nun nach der Stadt „Kabul“ und ansehen uns an, wie die Abfrage ausgeführt wird.

```
Explain select * from city
where name = 'Kabul'
;
```

[Data Output](#) [Explain](#) [Notifications](#)

	QUERY PLAN text
1	Seq Scan on city (cost=0.00..82.99 rows=1 width=31)
2	Filter: (name = 'Kabul'::text)

Da auf der Spalte *name* kein Index angelegt ist, erfolgt die Abfrage wie erwartet über einen Tabellenscan. Als nächstes generieren wir nun den Index auf diese Spalte und schauen uns wieder an was passiert, indem wir die oben aufgeführte Abfrage wieder anwenden.

```
Create index City_name
on City
(name)
;
```

Data Output Explain Notifications

QUERY PLAN	
text	🔒
1	Index Scan using city_name on city (cost=0.28..8.30 rows=1 width=31)
2	Index Cond: (name = 'Kabul'::text)

Nun wurde der von uns angelegte Index verwendet und die "Total Cost" sind auf 1/10 (8.30) gesunken. Verändern wir aber die Abfrage nur minimal und wollen alle Städte wissen welche mit „Kaf“ beginnen, entscheidet sich der Optimizer wieder einen Tabellenscan durchzuführen.

```
select * from city
where name like 'Kaf%'
;
```

Data Output Explain Notifications

QUERY PLAN	
text	🔒
1	Seq Scan on city (cost=0.00..82.99 rows=40 width=31)
2	Filter: (name ~~ 'Kaf%'::text)

Zum Schluss dieses Kapitels sei noch kurz erwähnt, dass falls wir einen Index anlegen und dieser nicht verwendet wird, können wir die Statistiken, auf die der Optimizer zugreift, noch aktualisieren lassen. Dies erfolgt mit nachfolgendem Befehl.

```
analyze city
;
```

Dieser Befehl sammelt die statistischen Daten über den Inhalt der Tabellen der Datenbank und speichert diese in der Tabelle „pg_statistic“ ab. Diese Statistiken helfen dem Optimizer den effektivsten Plan für eine Abfrage zu finden.

10 Schlusswort

Sie liebe Studierende haben nun mit diesem Skript und den ergänzenden Folien der Vorlesung Datenbanken I ein Werkzeug an der Hand sich in die Grundlagen von Datenbanken systematisch einzuarbeiten. Die von mir im Literaturverzeichnis angeführte Literatur und hier vor allem das sehr umfängliche und detaillierte Buch von Alfons Kemper und André Eickel können Ihnen dazu dienen sich noch tiefer mit der Materie zu beschäftigen. Aber noch so viele Bücher werden Ihnen nicht die Praxis ersetzen in der Sie die theoretischen Kenntnisse auch austesten müssen. Nach diesen zwei Semestern Grundlagen von Datenbanken müssten Sie aus meiner Sicht in der Lage sein die wichtigsten Vorgänge in einem DBMS nachvollziehen zu können.

Ich wünsche Ihnen dazu viel Spaß und Erfolg.

Gruß

Wolfgang Stark

11 Literaturverzeichnis

[Adam16]

Ralf Adams (2016)

SQL Der Grundkurs für Ausbildung und Praxis 2. Aufl.

Carl Hanser Verlag München ISBN: 978-3-446-45074-5

[Balz09]

Helmut Balzert (2009)

Lehrbuch der Softwaretechnik - Basiskonzepte und Requirements Engineering 3. Aufl.

2009 Spektrum Akademischer Verlag Heidelberg

ISBN: 978-3-8274-1705-3

[Gebh17]

Prof. Dr. Karl Friedrich Gebhardt (2017)

Vorlesungsskript Datenbanksystemen

[ElNa05]

Ramez Elmasri, Shamkant B. Navathe (2005)

Grundlage von Datenbanksystemen, Ausgabe Grundstudium 3. Aufl.

Pearson Education Deutschlang München

ISBN: 3-82737153-8

[KeEi15]

Alfons Kemper, André Eickel (2015)

Datenbanksystem, Eine Einführung 10. Aufl.

De Gruyter Verlag Oldenbourg

ISBN: 978-3-11-044375-2

[Mart81]

Janes Martin (1981)

Einführung in die Datenbanktechnik 1. Aufl.

Carl Hanser Verlag München

ISBN: 3-446-12929-4

[Piep11]

Lothar Piepmeyer (2011)

Grundkurs Datenbanksysteme 1. Aufl.

Carl Hanser Verlag München

ISBN: 978-3-446-42353-1

[Rupp07]

Chris Rupp & die SOPHISTen (2007)

Requirements-Engineering und –Management 4. Aufl.

Carl Hanser Verlag München Wien

ISBN: 10-446-40509-7

[SOPH08]

SOPHIST GROUP / Chris Rupp (2008)
Systemanalyse kompakt 2. Aufl.
Springer-Verlag Berlin Heidelberg – Spektrum Akademischer Verlag
ISBN:978-3-8274-1936-1

[Stud16]

Thomas Studer
Relationale Datenbanken: Von den theoretischen Grundlagen zu Anwendungen mit PostgreSQL
Springer, 2016
ISBN 978-3-662-46570-7

[Wirt91]

Niklaus Wirth (1991)
Algorithmen und Datenstrukturen
Vieweg+Teubner Verlag; Auflage: 3

[Zeit91]

Edgar Zeit (1991)
Programmierung des OS/2 Extended Edition Database Manager.
Friedrich Vieweg & Sohn Verlagsgesellschaft mbH, Braunschweig
ISBN: 3-528-04776-3

12 Links und Referenzen

12.1 Links

- [1] <https://www.youtube.com/channel/UCC9zrtAkl6yY4dpcnWrCHjA>
- [2] <https://www.postgresql.org/docs/11/index.html>

12.2 Datentypen

Diese Tabelle erhebt nicht den Anspruch der Vollständigkeit. Dazu lesen bitte im Online-Manual von PostgreSQL [2].

Datentyp	Speichergröße	Wertebereich / Beschreibung
integer, int	4 Bytes	Ganzzahlwert zwischen -2147483648 to +2147483647 (es gibt auch <i>smallint</i> , <i>bigint</i>)
decimal(p,s) numeric(p,s)	variabel	Zahl mit vorgegebener Vor- und Nachkommastellen (p Ziffern und davon s nach dem Komma)
real	4 Bytes	6 Dezimalzahlen (ungenau)
serial	4 Bytes	Ganzzahlwert mit Autoinkrement (integer). Es gibt auch <i>smallserial</i> und <i>bigserial</i>
character varying(n), varchar(n)		Zeichenkette mit der Länge n, gegebenenfalls wird sie mit Leerzeichen aufgefüllt
character(n), char(n)		Zeichenkette mit maximaler Länge n, gegebenenfalls wird sie mit Leerzeichen aufgefüllt
text		Zeichenkette mit beliebiger Länge
date	4 Bytes	Datum (2002-03-14)
time	8 Bytes	Tageszeit (10:38:20)
timestamp		Zeitstempel ('2001-09-28 03:00:00')
bytea		Für große Objekte (variabel großer binärer String)
now()		Aktuelles Datum mit Zeit(Zeitzone: "2019-02-13 20:22:19.528635+01")
boolean	1 Byte	Zustand: wahr oder falsch

13 Index

I

(min, max)-Notation 43

A

Abbildungsfunktionen 17
 Abfrageoptimierer 111
 abgeleitetes Attribut 27
 Abhängigkeitsbewahrung 56
 ACID 98
 ACID-Prinzip 15
 Aggregation 33
 Änderungsanomalie 48
 Anforderungsanalyse 18, 20
 ANY 87
 ANY,ALL,EXISTS,IN 87
 Attribute 25
 AUTO-COMMIT Modus 97
 Autoincrement 88

B

B-Bäume 108
 BEGIN {TRANSACTION} 97
 BETWEEN 84
 Beziehungstypen 23

C

CASCADE 69
 Chen-Notation 28
 CODASYL 12
 Constraints 62
 CREATE TABLE 62, 110
 CURRVAL 100

D

Data Retrieval Language 82
 Data-Definition Language 18
 Datenbankausprägung 15
 Datenbank-Definition 18
 Datenbankmodell 15
 Datenbankschema 15
 Datenbankserver 60
 Datenbanksprache 16
 Datenbankverarbeitungssystem 60
 Datenmodell 15
 Datenorientierteren Techniken 21
 Datentypen 63

Datenunabhängigkeit 17
 Datenunabhängigkeit
 logische Datenunabhängigkeit 17
 physische Datenunabhängigkeit 17
 Datenverarbeitungssystem 10
 deklarative Programmiersprache 58
 DELETE 68
 der logische Entwurf 36
 Determinante 49, 55
 Differenz 74
 dirty read 104, 105
 DISTINCT 83
 DMBS 8
 Duplikatentfernung 73
 Durchschnitt 77

E

E.F. Codd 13
 Einfügeanomalie 47
 Entität 23
 Entitäts-Typ 23
 Entity-Relationship-Modell 23
 Entwurfsprozesses 18
 Equi-Join 80
 ER-Modell 22
 EXCEPT 89
 Existenzabhängigen Entitäten 34
 Explain 111
 Externen Ebene 16

F

FOREIGN KEY 65
 Formale Sprache 71
 Fremdschlüssel 41
 Funktionale Abhängigkeit 49
 Funktionen
 AVG 84
 COUNT 84
 SUM 84

G

Generalisierung 33
 GROUP BY 85

H

HAVING 85
 Hierarchisches Datenmodell 11

I

Implementierungsentwurf.....	18
IMS.....	11
Index.....	110
Indextabelle.....	107
Ingres	13
INNER JOIN	92
INSERT	67
Instanz	39
INSTEAD OF INSERT Trigger	96
Interferenzregeln.....	49
Interne Ebene	16
INTERSECT.....	89
Isolationsstufen	104

J

JOIN	
Äußenen Join.....	92
Cross-Join	91
Equi-Join	91
Natural-Join	91
Theta-Join	90

K

Kardinalitäten	28
kartesischen Produkts.....	36
Kataloge.....	10
Knoten	108
Konkurrenzenden Zugriffen	102
Konzeptionelle Ebene	16
Konzeptionelle Entwurf	19
Krähenfußnotation	29
Martin-Notation	29
Kreuzprodukt	75
Künstlicher Schlüssel	25

L

Lastenheft.....	20
Logische Entwurf	18
Log-Puffer	99
Löschanomalie.....	48
lost update.....	103

M

Mehrwegbäume	108
Meta-Daten	14
Min-Max-Notation.....	29
Multiversion Concurrency Control (MVCC)	105

N

Natürlichen Join	80
Nebenläufigkeit	103
Netzwerdatenmodell.....	12
non repeatable read	104
Normalisierung	47
NoSQL-Datenbanken.....	15
n-stellige Beziehungen	31
NULL-Wert.....	38

O

Optimizer	111
-----------------	-----

P

P. P. Chen	22
Partielle Funktion	31
Partitionierung	33
Persistente Datenhaltung.....	11
pg_statistic	113
pgAdmin	60
PG-Indexes	111
Phantomproblem	104
Physikalische Entwurf.....	18
PostgreSQL	61
PostgreSQL.....	59
Prädikat	82
Primärschlüssel	26, 51
Primattribut	
prim.....	51
Projektion	48, 72

R

read only views	94
Recovery.....	15
Referentielle Integrität.....	46
Rekursive Beziehung	32
Relationale Datenbankmanagementsystem	9
Relationalen Algebra	36
Relationen-Schema	38
Relationship	23
RESTRICT	69
Revorevery	97

S

Schema	39
Schlüsselattribut.....	38
Schlüsselattribute	25
Schlüsselkandidaten	26, 51
Schwacher Entität	40
SELECT	82
Selektion	71
Semantische Integritätsbedingungen	46
SERIAL	88

SERIALIZABLE	104
Sicht (siehe View)	94
Sichten	16, 34
Spezialisierung	32
SQL-Schema	62
subquery	87
Superschlüssel	50
System R	13

T

Tabellenscan	110
Ternäre Beziehung	31
Theta-Join	79, 90
Transaktion	15, 97
Transitive Abhängigkeiten	53
Triviale funktionale Abhängigkeit	49
Tupel	36, 37

U

Umbenennung von Attributen	76
UNION	89

UPDATE	68
UPDATE ON	70

V

verbundtreu	56
Vereinigung	73
View	94

W

Wertebereich	24
Wissen	9
Wurzelement	12

Z

zeitinvariant	26
zusammengesetzten Attributen	27