Datenbanksysteme 1

Prof. Dr. Klaus Küspert

Semester: WS 2009/10

Vorwort

Dieses Dokument wurde als Skript für die auf der Titelseite genannte Vorlesung erstellt und wird jetzt im Rahmen des Projekts „Vorlesungsskripte der Fakultät für Mathematik und Informatik“ weiter betreut. Das Dokument wurde nach bestem Wissen und Gewissen angefertigt. Dennoch garantiert weder der auf der Titelseite genannte Dozent, die Personen, die an dem Dokument mitgewirkt haben, noch die Mitglieder des Projekts für dessen Fehlerfreiheit. Für etwaige Fehler und dessen Folgen wird von keiner der genannten Personen eine Haftung übernommen. Es steht jeder Person frei, dieses Dokument zu lesen, zu verändern oder auf anderen Medien verfügbar zu machen, solange ein Verweis auf die Internetadresse des Projekts `http://uni-skripte.lug-jena.de/` *enthalten ist.*

Diese Ausgabe trägt die Versionsnummer 3065 und ist vom 11. Juli 2010. Eine neue Ausgabe könnte auf der Webseite des Projekts verfügbar sein.

Jeder ist dazu aufgerufen, Verbesserungen, Erweiterungen und Fehlerkorrekturen für das Skript einzureichen bzw. zu melden oder diese selbst einzupflegen – einfach eine E-Mail an die Mailingliste `<uni-skripte@lug-jena.de>` *senden. Weitere Informationen sind unter der oben genannten Internetadresse verfügbar.*

Hiermit möchten wir allen Personen, die an diesem Skript mitgewirkt haben, vielmals danken:

- *Jens Kubieziel* `<jens@kubieziel.de>` *(2010)*

Inhaltsverzeichnis

Auflistung der Theoreme

Definitionen und Festlegungen

Teil I.

Datenbanksysteme 1

Vorbemerkung

Das Dokument spiegelt die Vorlesung zu Datenbanksystemen 1 und 2 wider. Es entspricht *nicht* der von Herrn Küspert gehaltenen Vorlesung, sondern stellt vielmehr eine Zusammenstellung diverser Fakten aus der Vorlesung dar. Daher können hier Dinge fehlen oder auch andere Fakten erwähnt sein. Wenn ihr also das Skript zufällig im Web gefunden habt und für die Vorlesung lernen wollt, haltet euch an die Homepage des Lehrstuhls für Datenbanken und Informationssysteme.

1. Einleitung

Die Vorlesung nimmt eine grundlegende Einführung in Datenbanksysteme vor. Nach einer Einführung wird zuerst das Entity-Relationship-Modell vorgestellt und gezeigt, wie die „Wirklichkeit“ in solch einem Modell abgebildet werden kann. Danach lernen wir diverse Datenbanksysteme kennen. Am Anfang steht das hierarchische Modell. Es erfolgt eine Vorstellung der Theorie zu den Modell und später praktizieren wir das Erlernte an dem System IMS. Auf das Netzwerk-Datenbankmodell folgt schließlich eine Einführung in das relationale Modell. Dort wird die Relationenalgebra sowie das -kalkül vorgestellt. Am Ende der Vorlesung zu Datenbanksysteme 1 erfolgt eine Einführung in SQL.

1.1. Anforderung an Datenbanksysteme

Zu Anfang wollen wir uns die Frage stellen, ob wir Datenbanksysteme überhaupt brauchen. Haben diese einen Nutzen, sind das nur Konstrukte der Theorie oder gibt es Probleme, die sich damit lösen lassen? Schließlich könnte man auf die Idee kommen, alle seine Daten auf einem Datenträger abzulegen. Das heißt, diese auf der Festplatte, USB-Stick oder ähnlichem zu speichern. Von dort werden die Daten dann wieder abgerufen und alles ist gut.

Bei großen Datenmengen und Zugriff von verschiedenen Seiten ist schnell klar, dass dieser Weg nicht korrekt sein kann. Man stelle sich eine große Firma vor, die deren Mitarbeiterstammdaten verwaltet. Für jeden Mitarbeiter wird eine Datei angelegt. Schon bei einer geringen Zahl an Mitarbeitern wird die Suche nach dem korrekten Datensatz problematisch. Insbesondere wenn mehrere den gleichen Namen besitzen. Weiterhin entstehen Probleme, wenn mehrere Kollegen auf denselben Datensatz schreibend zugreifen. Diese Probleme erweitern sich schnell, wenn die Datensätze ausgewertet werden sollen, d. h. wie ist das Durchschnittsgehalt in der Abteilung oder der Firma, wer wurde nach September 1996 eingestellt etc.

Daher kann eine solche Datenhaltung nicht (in allen Fällen) der richtige Weg sein. Welche Anforderungen sollte man an ein Datenbanksystem stellen?

Persistente Datenhaltung Nach dem Abschluss einer Transaktion (Speichern) bleibt das Ergebnis dauerhaft erhalten.

Große Datenmenge Das Datenbanksystem soll mit beliebig großen Datenmengen (Giga-, Tera-, Petabyte etc.) umgehen können.

Hohe Verfügbarkeit Datenbestände sollen 24/7 abruf- und bearbeitbar sein. Gerade bei großen Webseiten gibt es keine Wartungsfenster mehr. Vielmehr muss diese rund um die Uhr online sein.

Flexibilität Das Datenbanksystem muss in verschiedener Weise Flexibilität garantieren können. Dazu gehört, dass Daten flexibel ausgewertet werden können.

- Welche Daten haben Eigenschaft x?
- Welche Daten sind größer als y?
- Welche Daten haben Eigenschaft x und sind größer als y?

Weiterhin sollen die Daten nicht notwendigerweise nur an einem Standort gespeichert werden. Vielmehr ist es sinnvoll, Datensätze zu verteilen und trotzdem eine Auswertbarkeit *aller* Daten zu haben. Große Anbieter benötigen die Möglichkeit der Lastverteilung. Das heißt, es wird von vielen Rechnern auf einem Datenbestand zugegriffen. Ein Beispiel sind die Server der Wikipedia.

Benutzerfreundlichkeit Die Datenabfrage soll so benutzerfreundlich wie möglich sein.

Sicherheit Der Punkt Sicherheit hat wieder mehrere Aspekte. Zum einen soll es Sicherheit vor Datenverlust geben. Also auch bei einem plötzlichen Ausfall des Servers sollen die Daten nicht komplett verloren sein. Weiterhin sollte kein Unbefugter auf Daten zugreifen oder diese gar ändern können. Insbesondere ist die Rechteverwaltung granular, d. h. man kann angeben, wer Daten nur lesen, schreiben etc. darf.

Datenbanksysteme sollen bezüglich aller oben genannter Punkte Lösungen bieten. Dabei spielt die Performance eine wesentliche Rolle. In der Praxis unterscheidet man zwischen OLTP- und OLAP-Betrieb. **OLTP** steht für **Online Transaction Processing**. Hier gibt es viele kurze Verarbeitungsvorgänge, viele Nutzer, die zur selben Zeit arbeiten und schnelle Antwortzeiten. Dies wird überwiegend unser Thema sein. Hingegen steht **OLAP** für **Online Analytical Processing** und hat typischerweise komplexe Verarbeitungsvorgänge, wenige Nutzer und unkritische Antwortzeiten.

1.2. Begriffe

Im folgenden sollen einige wichtige Begriffe für die Vorlesung definiert werden. Diese sind zentral für die Vorlesung und ziehen sich wie ein roter Faden durch die Unterlagen.

Definition 1.1 (Datenbank)
Ein strukturierte Sammlung von Daten bzw. Datensätzen wird als **Datenbank** bezeichnet.

Definition 1.2 (DBMS, DBVS)
Ein **Datenbankmanagementsystem** (**DBMS**) oder **-verwaltungssystem** (**DBVS**) verwaltet die Datenbestände, d. h. alle Zugriffe gehen *ausschließlich* über das DBMS. Es übt die Kontrolle über die Datenbestände aus.

Definition 1.3 (Integritätsbedingungen)
Integritätsbedingungen sind Bedingungen, die an die Daten gestellt werden. Die Aufgabe des DBMS ist es u. a. die Einhaltung dieser Bedingungen zu überwachen.

Bemerkung 1.1
Eine Datenbank kann neben den primären Daten weitere Informationen enthalten. Dazu gehören:

- Indexe (Zugriffspfade)
- Beschreibungsinformationen, d. h. Metadaten, wie Feldlängen, -namen, Informationen über die Zugangsberechtigung etc.
- Integritätsbedingungen
- Aktive Elemente, d. h. Aktionen, die in Abhängigkeit von bestimmten Ereignissen Änderungen auslösen.
- Programme bzw. gespeicherte Prozeduren (vom englischen Wort: stored procedures). Diese sind in der Datenbank hinterlegt und laufen bei Aufruf der Datenbank ab.

1.3. Datenverwaltung mit Dateisystem

Eine erste Idee besteht darin, alle Daten mit Hilfe des Dateisystems zu verwalten. Das bedeutet, man nutzt Dateien und Verzeichnisse zur Ablage. Dies lässt sich einfach umsetzen und anfangs sicher auch leicht bedienen. Als Speichermedium kann ein USB-Stick, eine Festplatte oder ähnliches dienen.

Die Daten können verschieden organisiert werden. Eine Möglichkeit wäre, alle Daten fortlaufend (sequentiell) in einer so genannten **sequentiellen Datei** zu speichern. Die Ordnung der Daten entsteht durch die Reihenfolge der Speicherung. Die Datenfelder können von fester oder variabler Länge sein. Im ersten Falle ist der Zugriff einfacher. Es kann an eine vorgegebene Stelle in der Datei gesprungen und die Daten gelesen werden. Bei variabler Länge muss die Datei bei jedem Lesevorgang sequentiell erfasst werden.

Die **indexsequentielle Datei** ist eine zweite Form der Speicherung. Die Datensätze befinden sich hierbei in den Blättern eines Baums oder es gibt Verweise von den Blättern zu den Datensätzen. In der Regel ist das als B^*-Baum implementiert. Die Methode wird als **Index Sequential Access Method** (**ISAM**) oder **Virtual Storage Access Method** (**VSAM**) bezeichnet.

Schließlich ist es möglich, eine **Hashdatei** einzusetzen. Ein Feld wird als Schlüsselfeld festgelegt. Die Werte des Schlüsselfeldes werden dann über eine Hashfunktion in Speicheradressen umgerechnet.

Insgesamt wirft eine Datenhaltung im Dateisystem diverse Probleme auf. Je nach gewählter Form gibt es keinen schnellen Zugriff auf Daten. Die Geschwindigkeit ist von der Speicherungsstruktur abhängig. Bei Änderung dessen sind Probleme zu erwarten. Weiterhin ist der Fall der Reorganisation der Daten zu hinterfragen. Was passiert, wenn die Hashdatei vergrößert werden muss? In der Regel erfolgt der Zugriff auf *ein* Dateisystem. Bei vielen Zugriffen wird der Rechner eventuell überlastet. Schwierigkeiten sind bei der Vergabe von Zugriffsberechtigungen zu erwarten. Eine Festlegung von Rechten ist pro Datei möglich. Aber oft reicht diese Granularität nicht.

Insgesamt ist die Datenhaltung im Dateisystem unbefriedigend. Viele Gesichtspunkte sprechen gegen diese Lösung. Es kann daher nur im Einzelfall Vorteile bringen.

1.4. Architekturen

Üblicherweise arbeiten verschiedene Personengruppen an einem Datenbanksystem mit. Es gibt Administratoren der Datenbank sowie des Betriebssystems, Benutzer und ggf. andere Rollen. Alle haben eine andere Sicht auf das Datenbanksystem. Daher bietet sich an, eine Systemstruktur aus Komponenten, Ebenen, Schnittstellen etc. zu erschaffen. Diese abstrahiert die Sichtweise und führt dadurch für die Benutzerrollen zu einer Verringerung der Komplexität. Diese Abstraktion wird als **Architektur** bezeichnet. Im folgenden wollen wir verschiedene Architekturen kennen lernen.

1.4.1. ANSI-SPARC-Architektur

Die **ANSI-SPARC-Architektur** wird manchmal als **Drei-Schema-Architektur** bezeichnet und ist ein abstraktes Design für ein DBMS. Der Namensgeber war ANSI-SPARC und steht für American National Standards Institute, Standards Planning And Requirements Committee. Der Vorschlag[1] stammt aus dem Jahr 1975 und enthält eine Aufteilung in drei Ebenen.

Externe Ebene stellt Benutzern individuelle Benutzersichten, wie Formulare, Listen etc. bereit

Konzeptionelle Ebene beschreibt, welche Daten gespeichert sind und wie deren Beziehung untereinander ist. Designziel ist hier eine vollständige und redundanzfreie Darstellung aller zu speichernden Informationen.

[1] ANSI/X3/SPARC Study Group on Data Base Management Systems: (1975), Interim Report. FDT, ACM SIGMOD bulletin. Volume 7, No. 2

Interne Ebene beschreibt, wie und wo die Daten gespeichert werden. Dies ist die physische Sicht der Datenbank auf den Computer. Das Designziel ist ein effizienter Zugriff auf die Informationen.

Der Vorteil dieser Architektur liegt in der physischen wie logischen Datenunabhängigkeit. Durch die Trennung der Ebenen kann beispielsweise das Speichermedium gewechselt werden ohne das dies Einfluss auf die konzeptionelle oder externe Ebene hätte. Weiterhin wirken sich Änderungen im Layout der Datenbank nicht auf Formulare oder andere Schnittstellen aus. Details sind in dem Buch „The ANSI/SPARC DBMS Model“ von Donald Andrew Jardine (siehe [8]) erläutert.

Ein **Datenbankadministrator** legt zusammen mit dem **Anwendungsadministrator** das konzeptionelle Schema sowie allein das interne Schema fest. Er kennt nicht unbedingt das externe Schema. Der **Anwendungsadministrator** legt mit dem Datenbankadministrator das externe Schema und allein das konzeptionelle Schema fest. Er kennt nicht unbedingt das interne Schema. Der Endnutzer kennt das externe Schema, aber keine weiteren Schemata.

1.4.2. DIAM

Der Begriff **DIAM** steht für **Data Independent Accessing Model** und wurde 1973 von Michael Senko für IBM entwickelt. Später gab es dann Erweiterungen dieses Modells. Nach dem ursprünglichen Ansatz sieht DIAM vier Ebenen vor:

entity set model logische und anwenderneutrale Datenstrukturen. Es werden Mengen von Objekten dargestellt und vom DBMS verwaltet.

string model logische Zugriffspfade (Indexe).

encoding model Speicherungsstrukturen, physischer Zugang. Das bezeichnet die Art, wie die Daten kodiert, d. h. physisch im Speicher dargestellt, sind. Die Felder eines Datensatzes sind fortlaufend gespeichert. Variable langen Einträgen geht ein Längenfeld von 2 Byte voraus.

physical device model Speicherzuordnungsstrukturen, d. h. Datenzuordnung zum Dateisystem, Festplatte etc.

wird das wirklich so geschrieben?

2. Datenmodellierung mit dem Entity-Relationship-Modell

Das Kapitel wird Begriffe wie Entity, Entitytyp, Wertebereich einführen. Weiterhin lernen wir die grafische Darstellungsform von Entity-Relationship-Modellen, die sogenannten Entity-Relationship-Diagramme, kennen. Diese Modelle kürzen wir durch E/R-Modell ab. Eingedeutscht könnte man auch von einem **Gegenstand-Beziehungs-Modell** sprechen. E/R-Modelle beschreiben einen Ausschnitt aus der realen Welt.

Die Anfänge machte P. P. Chen. Er gilt als der Erfinder der Entity-Relationship-Modellierung[1]. Diese Details sind heute in praktisch jedem Lehrbuch zu Datenbanksystemen aufgeführt.

Die Modellierung auf abstrakter Ebene ist sinnvoll. Denn solch ein Modell ist unabhängig von der späteren Umsetzung in einem konkreten DBMS bzw. einem Datenbankmodell. Gerade im Vergleich zu der relationalen Darstellung mittels Tabellen ist das E/R-Modell übersichtlicher und kann wegen der leichten Verständlichkeit als Gesprächsgrundlage dienen.

Beispiel 2.1
Die Abbildung 2.1 zeigt ein unvollständiges E/R-Diagramm. Es beschreibt eine Vorlesung, die von einem Professor gelesen wird. Der Professor empfiehlt den Studenten Bücher zur Lektüre. Dabei ist dem Professor ein Name, eine Telefonnummer und ein Fachgebiet zugeordnet. Andere Bestandteile des E/R-Diagramms haben bestimmte Eigenschaften.

Datenbanksysteme unterstützen in aller Regel das Entity-Relationship-Modell nicht direkt. Stattdessen muss eine Umsetzung vom E/R-Modell zum Datenbankmodell erfolgen.

2.1. Entitäten, Attribute und Schlüssel

Definition 2.1 (Entität)
Eine **Entität** (vom engl. *entity*) ist ein bestimmtes, wohl unterscheidbares Objekt.

[1] P. P. Chen: The Entity-Relationship-Model – Towards a Unified View of Data. ACM Transactions on Database Systems (TODS). Bd. 1, Nr. 1, 1976.

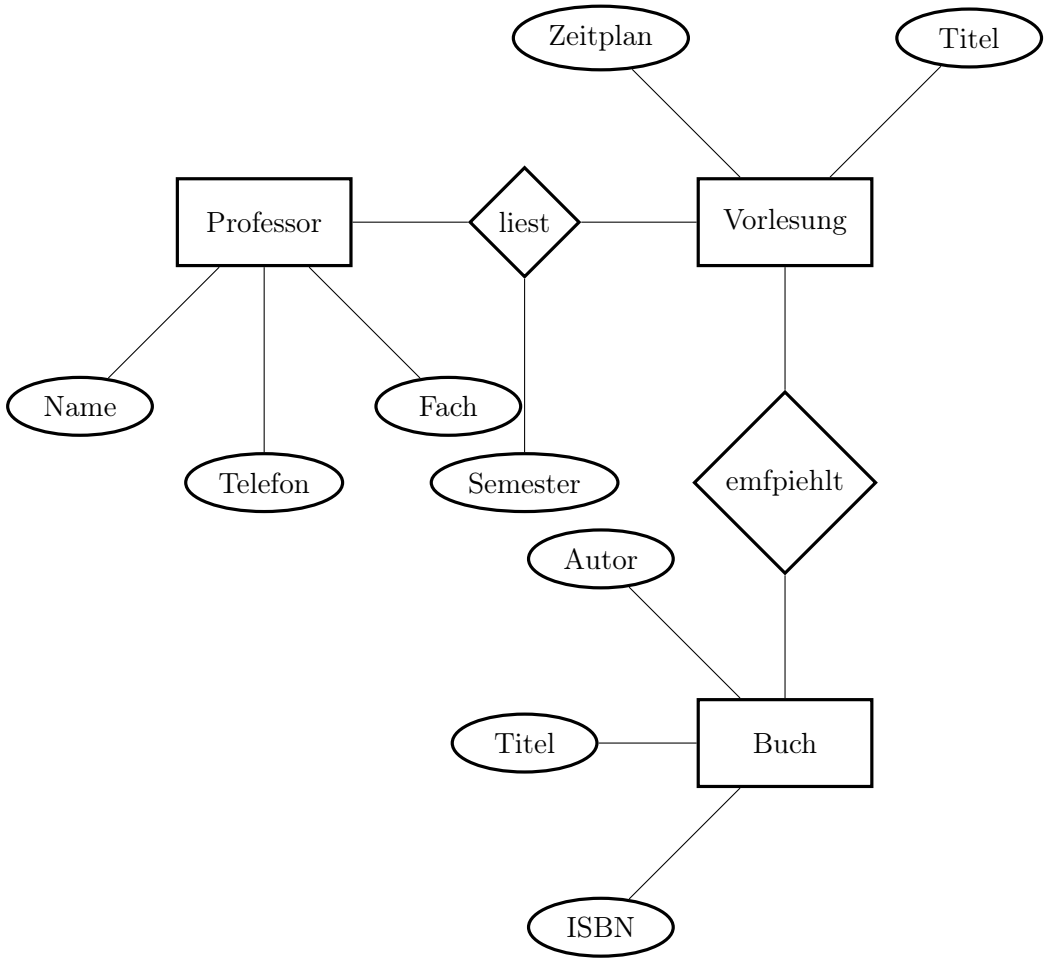


Abbildung 2.1.: Beispiel eines E/R-Diagramms

Definition 2.2 (Entitätstyp)
Ein **Entitätstyp** oder **Entitytyp** ist eine Zusammenfassung von Entitäten mit gleichen charakteristischen Merkmalen.

Beispiel 2.2
Die Angestellten Müller, Meier und Schulze sind Entitäten. Der Entitätstyp `ANGEST` steht für die Menge aller möglichen Angestellten. Weiterhin könnte es den Entitätstyp `GEBÄUDE`, `ABTEILUNG` etc. geben.

Bemerkung 2.1
Verschiedene Entitätstypen müssen nicht disjunkt sein. Vergleiche die Entitytypen `PERSON` und `STUDENT`.

Definition 2.3 (Attribut)
Ein **Attribut** ist eine Eigenschaft des Entitätstyps. Jedem Entitätstyp E wird eine nicht leere endliche Attributmenge A zugeordnet.

Beispiel 2.3
`E=ANGEST` und `A={Personalnummer, Name, Vorname, Gehalt}`

Allgemeiner lässt sich feststellen: $A = \{a_1, a_2, \ldots\}$. Dabei sind die a_i die Attribute und $i < \infty$.

Definition 2.4 (Wertebereich, Domäne, Domain)
Jedem Attribut $a \in A$ ist ein **Wertebereich** $\mathrm{dom}(a)$ zugeordnet. Dieser wird auch als **Domain** oder **Domäne** bezeichnet.

Beispiel 2.4
- $\mathrm{dom}(\text{Personalnummer}) = \mathbb{N}$
- $\mathrm{dom}(\text{Gehalt}) = \{\, g \in \mathbb{R} \mid 1000 \leq g \leq 10000 \,\}$

Die Festlegungen zum Wertebereich stellen einfache semantische Integritätsbedingungen dar. Auf der Entityebene (Ausprägungsebene) beschreiben die Attributwerte die Entität.

Definition 2.5 (Schlüssel)
Ein **Schlüssel** dient zur eindeutigen Identifizierung einer Entität e innerhalb des Entitätstyps E.

Beispiel 2.5
Sei `E=ANGEST` und `A={Personalnummer, Name, Vorname, Gehalt}`. Dann suchen wir eine Teilmenge $K \subseteq A$ so, dass die Attributwerte des Schlüssel eindeutig die Entität identifizieren. Beispielsweise könnte `K={Personalnummer}` sein. Dabei müssen wir voraussetzen, dass die Personalnummer innerhalb des Unternehmens eindeutig ist.

Bemerkung 2.2
Sollte im Beispiel 2.5 die Personalnummer nicht eindeutig sein, so könnte man ein weiteres Attribut hinzufügen. Allgemein wird die Minimalität eines Schlüssels gefordert, d. h. $\nexists K' \subset K$ mit der Eigenschaft, dass K' Schlüssel ist.

Sollte es mehrere Schlüssel zur Auswahl geben, so werden diese als Schlüsselkandidaten bezeichnet. Einer davon wird dann als **Primärschlüssel** gewählt. Üblicherweise wird dieser möglichst klein gewählt, d. h. er enthält wenig Attribute.

Primärschlüssel werden manchmal „künstlich“ gewählt. Meist existiert kein Schlüssel in natürlicher Weise oder er verliert diese Eigenschaft eventuell in der Zukunft.

2.2. Beziehungen

Definition 2.6 (Beziehung, Relation)
Eine **Beziehung** oder **Relation** ist eine Verknüpfung zwischen mindestens zwei Entitäten.

Beispiel 2.6
Wir nehmen an, dass es zwei Entitätstypen `ANGEST` und `PROJEKT` gibt. Dann besteht zwischen einem Angestellten a und einem Projekt p eine Beziehung, wenn a an p arbeitet. Zwischen Angestellten a_1 und a_2 besteht eine Beziehung, wenn a_1 Chef von a_2 ist.

Definition 2.7 (Beziehungstyp)
Ein **Beziehungstyp** E ist die Menge aller möglichen Beziehungen zwischen je einer Entität der beteiligten Entitätstypen E_i (mit $i \in \mathbb{N}$). Man kann R auch als kartesisches Produkt $R = E_1 \times E_2 \times \ldots \times E_n$ auffassen und als n-stelligen Beziehungstyp bezeichnen.

Bemerkung 2.3
Für die Ablage in der Datenbank sich die tatsächlich vorhandenen Entitätsmengen bzw. Beziehungsmengen interessant. Diese sind Teilmengen von E bzw. R.

Beispiel 2.7 (Beziehungstypen)
Man betrachtet in der Regel verschiedene Beziehungstypen. So hat jeder Angestellte eine Personalakte. Das heißt, die Entität Angestellter besitzt die Eigenschaft haben bezüglich der Entität Personalakte. Jeder hat genau eine. Also spricht man vom **1:1-Beziehungstyp**.

Die Angestellten arbeiten in verschiedenen Abteilungen. Herr Müller arbeitet in der Finanz- und der Steuerabteilung, Frau Petermann im Verkauf und Frau Lehmann in der Geschäftsführung sowie in der Organisation. Somit kann jeder Angestellte in mehreren Abteilungen arbeiten. Man spricht vom **1:n-Beziehungstyp**.

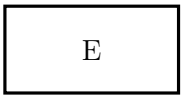
Schließlich arbeiten die Angestellten Müller, Meier und Schulze am Projekt Straßenbau, die Angestellten Müller und Petermann am Projekt Jahresabschluss sowie die Angestellten Fischer, Meier und Petermann am Projekt Einkaufsplanung. In dem Falle sind verschiedene Angestellte mit verschiedenen Projekten betraut und man spricht vom **n:m-Beziehungstyp**.

Bemerkung 2.4 (Integritätsbedingungen)
- Von jeder Beziehung r_i geht *genau eine* Kante aus.
- Nicht von jeder Entität e muss eine Kante ausgehen.

Definition 2.8 (Grad)
Die Zahl der an einer Beziehung beteiligten Entitätstypen wird als **Grad** bezeichnet.

2.3. Entity-Relationship-Diagramme

Die Entity-Relationship-Diagramme werden auch als E/R-Diagramme bezeichnet und sind die grafische Darstellung von Entitäts- und Beziehungstypen. Sie bestehen aus folgenden Elementen:

Entitätstyp 

Beziehungstyp 

Attribut 

Schlüsselattribut 

Zwingende Beziehung Sei $R = E \times E'$. Betrachte $e_1 \in E$. Dann gibt es stets mindestens ein $e_2 \in E'$ mit dem e_1 innerhalb von R in Beziehung steht, $r = (e_1, e_2)$.

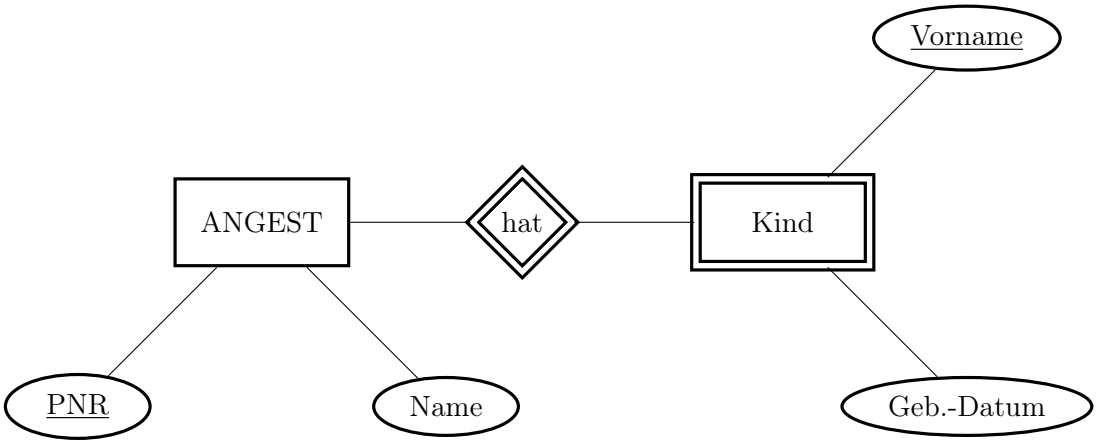
Optionale Beziehung Entity e_1 darf in E existieren, ohne dass es über R zu einer Entität $e_2 \in E'$ in Beziehung steht.

Definition 2.9 (Kardinalität, Komplexität)
Die **Kardinalität** oder **Komplexität** eines Beziehungstyps drückt aus, zu wie vielen Entitys $e_j \in E'$ ein Entity $e_i \in E$ in Beziehung stehen darf.

2.4. Schwache Entitytypen

Manche Entitäten existieren nur in Verbindung mit anderen Entitäten. Diese werden als **schwache Entität** bezeichnet. Anders gesagt muss zur eindeutigen Identifizierung einer Instanz eines schwachen Entitätstyps die Beziehung mit herangezogen werden. Also identifizieren bestimmte Attributwerte der Entity *plus* die Beziehung es eindeutig.

Beispiel 2.8
Manche Angestellte haben Kinder. Diese sollen ebenfalls mit in der Datenbank geführt werden. Diese Kinder „existieren“ nur zusammen mit den Eltern. Im E/R-Diagramm wird das wie folgt dargestellt:



Das heißt doppelte Linien beim Entity- und beim Beziehungstyp kennzeichnen schwache Entitätstypen.

2.5. Erweiterungen des E/R-Modells

2.5.1. Erweiterungen bei Attributen

Optionale Attribute Das Attribut existiert für den Entitätstyp bzw. den Beziehungstyp. Aber es muss nicht für jede Entität oder Beziehung einen definierten Wert annehmen. Grafisch wird das durch einen Kringel in der Verbindungslinie dargestellt. Optionale Attribute sind primär für den Fall gedacht, wo strukturelle Unterschiede zwischen den Entitys eines Typs erlaubt sind

Strukturierte Attribute setzt sich aus anderen Attributen zusammen. Die Komponenten eines strukturierten Attributs sind benannt und bezüglich des Wertebereichs meist inhomogen.

Mengenwertiges Attribut Der Attributwert ist eine Wertemenge, wie beispielsweise die Telefonnummern einer Person oder die Kinder des Angestellten. Im letzten Fall ist die Entscheidung, ob es sich um ein schwaches oder ein mengenwertiges Attribut handelt regelmäßig nicht einfach.

Virtuelle Attribute sind nicht real gespeichert. Die Werte werden durch eine Berechnungsvorschrift ermittelt. Die virtuellen Attribute kennzeichnet man durch gestrichelte Linien.

2.5.2. Erweiterung um Spezialisierung und Generalisierung

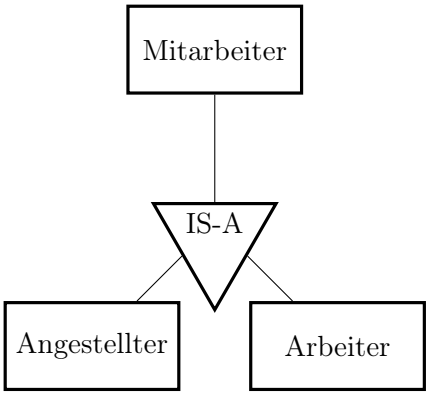
Spezialisierung und Generalisierung sind Bestandteil von erweiterten E/R-Modellen. Bei der **Generalisierung** werden Eigenschaften ähnlichen Entitytypen ermittelt und einem

gemeinsamen Obertyp zugeordnet. Die ähnlichen Entitytypen heißen **Untertypen** des Obertyps. In der umgekehrten Richtung sind die Untertypen eine **Spezialisierung** des Obertyps.

Beispiel 2.9
Jeder Angestellte und jeder Arbeiter ist ein Mitarbeiter. Beim Mitarbeiter handelt es sich damit um eine Generalisierung und Angestellte sowie Arbeiter sind die Spezialisierung. Die dabei entstehende Beziehung wird als **IS-A-Beziehung** bezeichnet. Das bedeutet, ein Angestellter *ist ein* Mitarbeiter etc.

Weiterhin kann man sagen, dass Mitarbeiter ein **Supertyp** von Angestellten und Arbeitern ist und die beiden letztgenannten sind ein **Subtyp** von Mitarbeiter.

Diese Beziehung stellt man im E/R-Diagramm wie folgt dar:



3. Das hierarchische Datenbankmodell

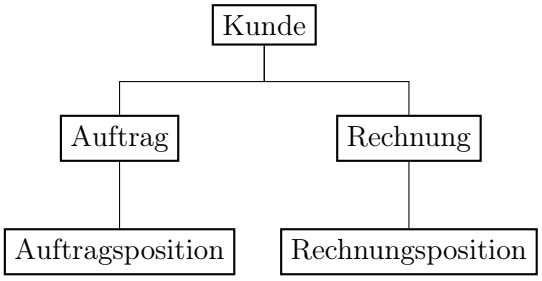
In diesem Kapitel wollen wir die Möglichkeiten der Modellierung mit dem hierarchischen Datenbankmodell (HDM) kennen lernen und einen Zusammenhang zwischen dem Modell und dem E/R-Modell herstellen. Im weiteren Verlauf werden die reinen Hierarchien in Richtung vernetzter Strukturen erweitert. Dabei bedienen wir uns sogenannter virtueller Satztypen. Als Beispiel für ein hierarchisches Datenbanksystem wird uns IMS dienen.

Im Allgemeinen sollte am Ende des Kapitels verstanden werden, was die Möglichkeiten eines HDM sind, ohne dass jemand mit einem spezifischen DBMS umgehen kann. Dazu wären weitergehende Details nötig. Weiterhin existiert eine Vielzahl von DBMS, die auf das hierarchische Modell setzen. Alle lassen sich in unterschiedlicher Weise bedienen.

Definition 3.1 (Hierarchisches Datenbankmodell)
Das **hierarchische Datenbankmodell** bildet die reale Welt durch eine hierarchisch aufgebaute Baumstruktur ab.

Bemerkung 3.1
Es muss sich nicht notwendigerweise um eine Hierarchie handeln. Vielmehr kann es mehrere geben. Dabei darf jeder Entitytyp nur einmal vorkommen. Innerhalb des Datenbankschemas muss der Name des Entitytyps eindeutig sein.

Beispiel 3.1
Eine Miniwelt mit Kunden und Aufträgen ließe sich mit folgender Baumstruktur modellieren.

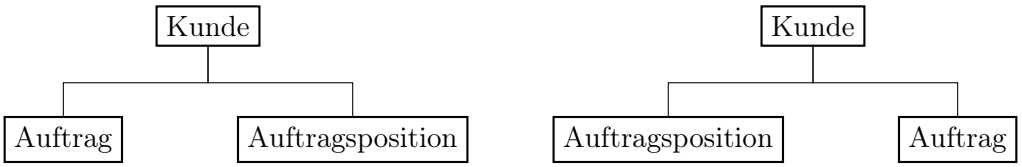


Diese Darstellung entspricht der **Typebene**. Wenn die Einträge mit konkreten Werten gefüllt werden, spricht man von der **Ausprägungsebene** bzw. **Instanzenebene** oder **Satzebene**.

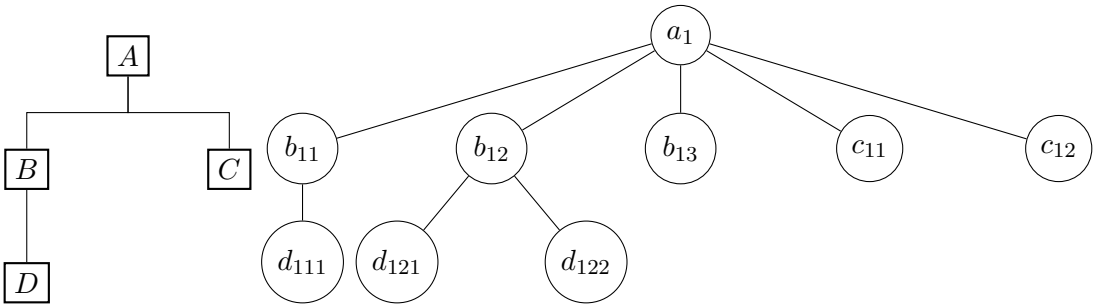
3.1. Bestandteile einer hierarchischen Datenbank

Definition 3.2 (Hierarchieordnung)
Eine **Hierarchieordnung** drückt aus, dass die Reihenfolge des Auftretens von Kindern eines Vaterknotens von Bedeutung ist.

Beispiel 3.2
Die beiden unten stehenden Hierarchieordnungen sind nicht identisch. Denn die Reihenfolge in der Typebene bestimmt die Reihenfolge der Abarbeitung auf der Instanzebene.



Sei im konkreten Beispiel folgende Hierarchie gegeben. Dabei ist sowohl Typ- wie auch Instanzebene gezeichnet.



Die Abarbeitung erfolgt in Präorder-Reihenfolge, d. h. $a_1 \to b_{11} \to c_{111} \to b_{12} \to d_{121} \to d_{122} \to b_{13} \to c_{11} \to c_{12}$.

Das Datenbankschema besteht aus:

- einer Menge von (benannten) Entitytypen
- einer Menge von (benannten) Hierarchien über den Entitytypen (versehen mit einer Hierarchieordnung)

Die Daten bestehen aus:

- einer Menge von Entitäten. Diese können auf der Ausprägungsebene in Beziehung stehen.
- einer Ordnungsreihenfolge innerhalb der Entitymenge.

Bemerkung 3.2 (Eigenschaften/Bezeichnungen)
- Jeder Entitytyp einer HDM gehört zu genau einer Hierarchie.
- Der oberste Entitytyp wird als **Wurzeltyp** bezeichnet.

- Für den Wurzeltyp existiert ein **Primärschlüssel**. Dieser erlaubt die eindeutige Identifizierung einer Entität aus der zugehörigen Entitymenge. Das bedeutet, es gibt einen Einstiegspunkt zum zugehörigen Baum.
- Jede Entität einer hierarchischen Datenbank ist auf einem der folgenden Wege erreichbar:
  - Wurzelentitäten über den Wert des Primärschlüssels
  - Reihenfolge der Ordnung auf Entity-Set-Ebene
  - Entitäten, die keine Wurzeln sind, über die Vaterentität

Damit ergeben sich diverse Konsequenzen. Zum einen ist ein direkter Zugriff auf eine beliebige Entität auf der Ausprägungsebene im HDM *nicht* möglich. Vielmehr ist Verarbeitung einer Speicherung auf Dateiebene, wie wir sie in Abschnitt 1.3 diskutiert haben, sehr ähnlich. Wobei die rein sequentielle Dateiorganisation gar kein Hineinspringen erlaubt. Von daher ist der Einsprungspunkt auf Wurzelebene bereits ein Vorteil.

Im hierarchischen Datenbankmodell wird keine Operation benötigt, die zum Vater eines Kotens geht. Denn wegen der oben beschriebenen Vorgehensweise ist der Vater immer bekannt. Das in Kapitel 4 besprochene Netzwerk-Datenbankmodell erlaubt beliebige Einstiegspunkte und benötigt daher die obige Operation.

3.2. Erweiterung des hierarchischen Datenbankmodells

Der Entitytyp müsste eigentlich mehrfach, also in verschiedenen Hierarchien auftreten. Dies ist in dem Modell jedoch verboten. Erweiterungen versuchen, den Konflikt zu lösen.

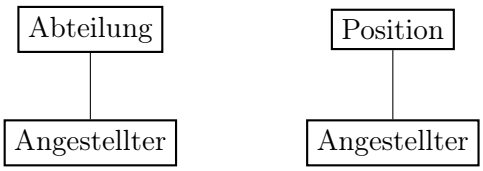
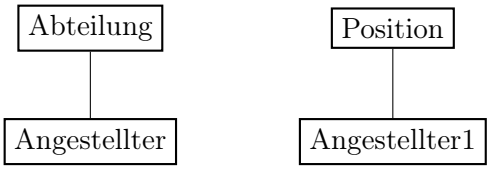


Abbildung 3.1.: Mehrfacher Entitytyp

3.2.1. Einführung von Redundanz

Man könnte verschiedene Hierarchien erlauben, in denen Entitytypen mehrfach vorkommen.



Aus Sicht des Datenbanksystems sind Angestellter und Angestellter1 unterschiedliche Entitytypen. Das heißt, dass die Entitäten aus der realen Welt in der Modellierung zweimal vorhanden sind. Dies ist eine Diskrepanz, die das Datenbanksystem nicht überwachen kann. Die Kontrolle muss durch den Benutzer geschehen. Letztlich ist dies nicht akzeptabel und daher abzulehnen.

3.2.2. Einführung von virtuellen Entitytypen

Hierbei existieren die „doppelten“ Entitäten in nur einer Hierarchie physisch. Aus anderen Hierarchien wird auf diese Entität per Zeiger verwiesen. Benutzer sehen diese Realisierung nicht. Das Datenbanksystem kann in diesem Modell wieder seine Stärken ausspielen. Die Abbildung 3.2 zeigt eine Ausprägung eines virtuellen Entitytyps. Analog könnte der virtuelle Angestellte sich auch links befinden.

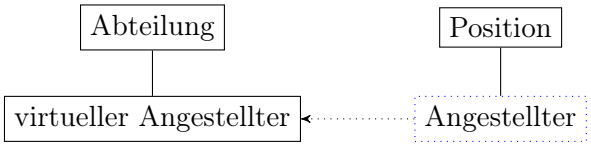
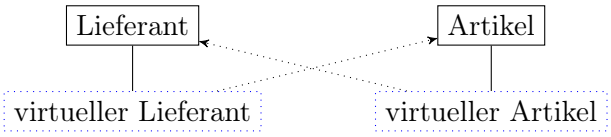


Abbildung 3.2.: Virtuelle Entitytypen

Mit dieser Konstruktion lassen sich alle Angestellten einer bestimmten Abteilung oder einer bestimmten Position effizient auffinden. Jedoch ist die Position oder Abteilung eines bestimmten Angestellten ineffizient zu finden.

Nun sei folgende n: m-Beziehung gegeben und wir suchen eine Repräsentation über ein hierarchisches Modell bei dem der Einstieg sowohl über den Lieferanten wie auch über den Artikel erfolgen kann. Das heißt, beide sollen gleichberechtigt dargestellt werden.

Die Lösung liegt hier darin, zwei virtuelle Entitytypen zu definieren. Diese verweisen jeweils auf den gegenüberliegenden Entitytypen.



Was passiert nun, wenn der Beziehungstyp „liefert“ Attribute, wie beispielsweise eine Menge, besitzt? Dann werden die Werte zusammen mit den Zeigern in der Ausprägungen der virtuellen Satztypen gespeichert. Also gibt es hierbei Redundanz.

3.3. IMS als Beispiel eines hierarchischen Datenbanksystems

IMS ist eine Abkürzung und steht für **Information Management System**. Es wurde zwischen 1966 und 1968 von IBM, Rockwell und Caterpillar für das Apollo-Mondprogramm entwickelt. Seit 1969 wird es von IBM bis heute weiterentwickelt.[1]

IMS speichert die Daten durch ein hierarchisches Modell. Dies wird durch Datenblöcke, die **Segment** genannt sind, realisiert. Jedes Segment kann verschiedene Daten enthalten. Die Teile bezeichnet man als **Feld**. Das Datenbankmodell wie auch die Sprache sind nicht sehr flexibel und schwer zu beherrschen. Nichtsdestotrotz wird das Datenbanksystem weltweit oft eingesetzt und gehört zu den umsatzstärksten Produkten des Konzerns.

3.3.1. Begriffe und Eigenschaften

Definition 3.3 (Segment)
Ein Knoten in der Typhierarchie wurde bisher als Entitytyp bezeichnet. Innerhalb von IMS heißt der **Segment**. Ein Segment besteht in der Implementation aus Metaangaben (Typecode, Pointer etc.) und den eigentlichen Anwendungsdaten.

Definition 3.4 (Feld)
Jedes Segment besteht aus einem oder mehreren **Feldern**.

Bemerkung 3.3
Die Segmente der Typhierarchie werden im Sinne der Präorder-Reihenfolge als geordnet betrachtet.

Definition 3.5 (Wurzelsegment)
Es gibt ein unabhängiges, ausgezeichnetes **Wurzelsegment**. Alle anderen Segmente sind **abhängige Segmente**.

Bemerkung 3.4
Man spricht auch vom **Vatersegment**, **Kindsegment** sowie vom **Geschwistersegment**.

[1] Tiefergehende Informationen zu IMS ist im IBM Redbook unter [11].

Definition 3.6 (Satz)
Die Knoten der Ausprägungsebene trugen bislang den Namen Entität. In IMS werden sie als **Satz** oder **Record** bezeichnet. Die Sätze enthalten eine **Feldausprägung**.

Bemerkung 3.5
Übertragen auf die Typebene gibt es also einen **Wurzelsatz**, **abhängige Sätze**, **Elternsatz**, **Kindsatz** und **Geschwistersatz**.

Es sind keine „Waisenkinder“ erlaubt, d. h. ein Kindsatz kann nur solange existieren, wie es einen zugehörigen Elternsatz gibt. Weiterhin erfolgt die Zuordnung von Kindrecord zum Elternrecord durch den Anwendungsprogrammierer. Das ist auch die schlechte Nachricht: Die Konsistenzwahrung erfolgt durch ein Anwendungsprogramm!

3.3.2. Schemadefinition im IMS

Konzeptionelles Schema

Die Struktur einer IMS-Datenbank, also die Typhierarchie, wird als **Physical Database Record Type** bezeichnet und wird durch eine **Database Definition** (**DBD**) festgelegt. Die DBD beschreibt einen wesentlichen Teil des konzeptionellen Schemas der IMS-Datenbank. Daneben kommen noch Aspekte des internen Schemas hinzu. Das heißt, IMS bietet keine klare Trennung zwischen der Beschreibung der konzeptionellen und der internen Ebene. In der DBD können nur 1:n-Beziehungstypen modelliert werden.

Beispiel 3.3
Nun sei eine Datenbank mit Informationen zu Kursen (Seminar, Schulung) gegeben. Das erste Segment enthält Informationen zu dem Namen des Kurses sowie dessen Nummer. Weiterhin wird modelliert, welche Voraussetzungen zur Teilnahme an einem Kurs vorliegen müssen sowie wie ein Angebot konkret aussieht. Bei letzterem ist die Angebotsnummer, das Datum und der Ort Teil des Segments. Unterhalb des Angebots finden sich spezielle Informationen zum Leiter des Kurses sowie zu den Teilnehmern. Eine Definition des Physical Database Record Type könnte so aussehen.

```
1  DBD     NAME=KursDB
2  SEGM    NAME=Kurs, BYTES=36
3  FIELD   NAME=(KursNr,SEQ), BYTES=3, START=1
4  FIELD   NAME=Titel, BYTES=33, START=4
5  SEGM    NAME=Vorauss, PARENT=Kurs, BYTES=3
6  FIELD   NAME=(VorNr,SEQ), BYTES=3, START=1
7  SEGM    NAME=Angebot, PARENT=Kurs, BYTES=21
8  FIELD   NAME=(AngNr,SEQ), BYTES=3, START=1
9  FIELD   NAME=DATUM, BYTES=6, START=4
10 FIELD   NAME=Ort, BYTES=12, START=10
11 SEGM    NAME=Kursleiter, PARENT=Angebot, BYTES=23
```

```
12 FIELD NAME=(PersNr,SEQ), BYTES=5, START=1
13 FIELD NAME=Name, BYTES=18, START=6
14 SEGM  NAME=Teilnehmer, PARENT=Angebot, BYTES=41
15 FIELD NAME=(TnNr,SEQ), BYTES=3 , START=1
16 FIELD NAME=Name, BYTES=18, START=4
17 FIELD NAME =Ort, BYTES=20, START=22
18 END
```

Listing 3.1: Definition des Physical Database Record Type

Die Reihenfolge der Segment-Definitionen (**SEGM**) bestimmt die Präorder-Reihenfolge der Segmente in der Typhierarchie. Alle Felder sind nicht typisiert und besitzen eine feste Länge. Die Angabe SEQ legt fest, dass die Segmentausprägungen nach diesen Feldwerten *aufsteigend* sortiert sind. Damit die Felder mit SEQ gleichzeitig Schlüsselfelder.

In **SEGM** erkennen wir die Segmente wieder und die nachgeordneten **FIELD**-Einträge sind die entsprechenden Felder.

Die ausführliche Variante einer solchen Definition ist bei IMS wesentlich komplizierter. Hier wurden nur die wichtigen Punkte zusammengetragen.

Externe Schemas

Auf IMS kann nicht ad hoc zugegriffen werden. Stattdessen erfolgt der Zugriff immer über ein Anwendungsprogramm. Diese Programme sehen nicht die physische Datenbank, sondern einen **Programm Communication Block** (**PCB**). Ein Programm kann auch mehrere PCBs verwenden. Diese bilden dann **Program Specification Block** (**PSB**), also das externe Schema.

Beispiel 3.4
Das folgende Listing zeigt den Program Specification Block zum vorigen Beispiel.

```
PCB      DBDNAME=KursDB, KEYLEN=9
SENSEG NAME=Kurs, PROCOPT=G
SENSEG NAME=Angebot, PARENT=Kurs, PROCOPT=G
SENFLD NAME=(AngNr,SEQ), START=1
SENFLD NAME=DATUM, START=4
SENSEG NAME=Teilnehmer, PARENT=Angebot, PROCOPT=GID
PSBGEN LANG=COBOL, PSBNAME=KursMaint
END
```

Die in der SENSEG-Anweisung aufgeführten Segmente werden sichtbar gemacht. Man spricht auch von sensibilisierten oder sensitiven Segmenten. Diese sind Teil des externen Schemas und damit durch das Programm ansprechbar. Weiterhin sensibilisiert SENFLD die entsprechenden Felder in den Segmenten. Sollte auf ein SENSEG kein SENFLD folgen, gelten alle Felder als sensibilisiert. PROCOPT steht für „processing options“ und definiert Zugriffsrechte. Dabei steht `G` für „get“, `I` für „insert“ und `D` für „delete“.

Zugriff mittels DL/I

Das **Data Language Interface**, manchmal als **Data Language/One** bezeichnet, wird als Sprache für den IMS-Datenbankzugriff benutzt. Zum besseren Verständnis betrachten wir die Abfolge bei einem lesenden Zugriff.

1. Die Anwendung ruft das Anschlussmodul mittels eines Unterprogrammaufrufs auf und übergibt die Anfrage in der Parameterliste.
2. Die Anfrage wird an das DBMS weitergegeben.
3. Das DBMS bearbeitet die Anfrage und stellt das Ergebnis bereit.
4. Ergebnis wird zurückgegeben.
5. Das Anwendungsprogramm analysiert den Rückgabewert und greift ggf. auf die Treffer zu.

Im folgenden lernen wir eine Auswahl des Operationsvorrats von DL/I kennen. Diese stellt nur eine grobe Übersicht dar. Für Details wird empfohlen, entsprechende Lektüre zu konsultieren.

`GET UNIQUE (GU)` direktes Positionieren auf einen Record und Lesen einer Segmentausprägung. Einstieg erfolgt von außen

`GET NEXT (GN)` Zugriff auf nächste Segmentausprägung ausgehend von aktueller Position

`GET NEXT WITH PARENT (GNP)` wie oben nur mit Vater-Segment-Ausprägung

`GET HOLD (GHU, GHN, GHNP)` wie bei obigen Varianten, nur die Segmentausprägung, auf die positioniert wurde, kann geändert werden.

`INSERT (INSRT)` Einfügen einer neuen Segmentausprägung

`DELETE (DLET)` Löschen einer Segmentausprägung

`REPLACE (REPL)` Ändern einer Segmentausprägung

4. Das Netzwerk-Datenbankmodell

In dem Kapitel wollen wir das Netzwerk-Datenbankmodell (NDBM) einordnen bzw. abgrenzen und die Mächtigkeit des Ansatzes studieren.

Das obige hierarchische Modell und speziell IMS sind proprietär. Das Netzwerk-Datenbankmodell wurde von Anfang an mit dem nicht proprietären Ansatz entwickelt. Es entstammt der **Conference on Data Systems Language** (**CODASYL**), weshalb es manchmal als CODASYL-Datenbankmodell bezeichnet wird. Aufgrund der Herkunft ist das Modell stark von COBOL beeinflusst. Ende der 60er Jahre wurde die **Data Definition Language** (**DDL**) und die **Data Modelling Language** (**DML**) spezifiziert. Der erste CODASYL-Report erschien im Jahr 1971. Auf der Basis eines sieben Jahre später erschienenen weiteren Vorschlages wurden diverse Systeme geschaffen. Dazu gehören das **Universelle Datenbanksystem** (**UDS**) von Fujitsu-Siemens bzw. **IDMS** von Computer Associates. Seit 1987 liegt ein ISO-Standard vor.

Für das Modell wurden drei Sprachen vorgeschlagen:

- Schema Data Description Language oder Schema-Datenbeschreibungssprache
- Subschema Data Description Language oder Subschema-Datenbeschreibungssprache
- Data Manipulation Language oder Datenmanipulationssprache

Die Sprachen sind vergleichsweise komplex und schwer zu erlernen. Im Vordergrund steht der navigierende Zugriff. Dieser erfolgt primär aus der Programmiersprache heraus.

Das Netzwerk-Datenbankmodell besitzt eine hohe Beschreibungsmächtigkeit und ist daher näher am E/R-Modell als die hierarchische Variante.

Das NDM sieht als eine Informationseinheit einen **Record** bzw. **Satz** vor. Records können aus mehreren atomaren Komponenten zusammengesetzt sein. Weiterhin lässt sich die Zusammengehörigkeit zwischen den Records darstellen. Dabei werden zwei Records miteinander verkettet und bilden eine **Menge**. Diese wird als **Set** bezeichnet. Ein Set kann man als **Sammlung** verstehen.

Zur Datenmodellierung werden zwei Konstrukte verwendet. Das sind **Satztypen**. Diese entsprechen den Entitytypen. Außerdem gibt es spezielle 1:n-Beziehungstypen, die **Set-Typen**. Die Beziehungstypen sind ausschließlich zweistellig und benannt. Man kann sich das als zyklischen gerichteten Graphen vorstellen. Eine Netzwerk-Datenbank besteht

aus einem Schema und so genannten Ausprägungen (vom englischen Wort „occurences“). Die Ausprägungen setzen sich aus **Set Occurrence** und **Record Occurrence** zusammen

4.1. Record-Typ

Definition 4.1 (Record-Typ)
Ein **Record-Typ** oder **Satztyp** T_R ist ein Tupel $(A_{R_1}, \ldots, A_{R_n})$. Dabei sind die A_{R_i} paarweise verschiedene Attributnamen und zu jedem A_{R_i} ist eine Menge D_i, die **Domäne** gegeben[1].

Definition 4.2 (Record)
Ein n-stelliger **Record** bzw. **Satz** vom Typ T_R ist ein Element des kartesischen Produkts $D_1 \times \ldots \times D_n$, also $(d_1, \ldots, d_n)$ mit $d_i \in D_i$ für $1 \leq i \leq n$[2].

4.2. Set-Typ

Definition 4.3 (Set-Typ)
Ein **Set-Typ** ist ein (zweistelliger) Beziehungstyp S zwischen den Record-Typen R und R' mit $R \neq R'$.

Definition 4.4 (Owner-, Member-Typ)
In der Definition 4.3 wird R als **Owner-Typ** und R' als **Member-Typ** bezeichnet.

Zwischen R und R' herrscht ein 1:n-Beziehungstyp. Dies wird nicht explizit in der grafischen Darstellung notiert.

Auf der Ausprägungsebene gilt:

- Zu *jedem* Record r aus dem Record-Typ R gehört eine Set Occurence s aus S. Diese darf leer sein.
- Jeder Record r' aus R' gehört zu maximal einer Set Occurence s aus S. Damit ist der Owner Record zu einem Member Record innerhalb eines Set immer eindeutig definiert. Waisenkinder sind erlaubt.

[1] Definition aus [3]
[2] Definition aus [3]

4.3. Bachman-Diagramme

Die Record- und Set-Typen besitzen ein hohes Detailreichtum. Daher ist es sinnvoll, diese in grafischer Form darzustellen. Im Laufe der Zeit hat sich eine Form durchgesetzt, die von der Ausprägungsebene der Records und Sets abstrahiert und nur Record- sowie Set-Typen abbildet. Diese Abbildungen werden nach dem amerikanischen Forscher Charles Bachmann (1924) als **Bachman-Diagramme** bezeichnet.

Ein Record-Typ wird hierbei als Rechteck mit der Typbenennung dargestellt. Attribute bleiben unberücksichtigt. Ein fiktiver Record-Typ ist als Kreis symbolisiert.

Die Set-Beziehung zweier Record-Typen ist durch einen Pfeil gekennzeichnet. Dieser geht vom Owner-Typ aus, endet beim Member-Typ und ist durch den Set-Typ-Namen markiert.

Weiteres schreiben

5. Das relationale Datenbankmodell

Die Datenbankmodelle, die wir bisher kennen gelernt haben, sind weit vom Endbenutzer entfernt. Die Datenbank ist nur von einem Anwendungsprogramm heraus ansprechbar. Also benötigt der Benutzer Programmierkenntnisse. Die Datenbanksprachen selbst sind schwer zu erlernen und kaum fehlerfrei anzuwenden. Somit ist die Benutzung nur einem kleinen Kreis von Spezialisten vorbehalten. Diese Eigenschaften führten dazu, dass Fehler gemacht wurden. Der modellierte Zustand entsprach nicht dem Zustand in der realen Welt und das DBMS wiederum konnte Inkonsistenzen nicht feststellen.

Daher sollte ein neues Datenbankmodell geschaffen werden. Dies soll eine mathematisch fundierte Grundlage und einfache Datenbanksprache besitzen. Der Benutzer beschreibt, was er machen möchte. Die Umsetzung sowie die Überwachung der Integrität und der Konsistenz wird dem DBMS überlassen. Dieses besitzt auch Informationen zur Semantik der Daten und kann die Korrektheit automatisch prüfen. Das Ergebnis dieser Überlegungen sind die relationalen Datenbanken.

Ab Mitte der 1970er Jahre gab es von IBM das System R als Prototyp sowie von der Universität of Berkeley die Datenbank Ingres. Aus System R wurde später DB2 und Ingres wurde unter dem Namen weiterentwickelt. Davon abgeleitet ist unter anderem die Freie Software Postgres. Weitere bekannte relationale Datenbanken sind Oracle, Informix, Sybase, Adabas D, MySQL und viele mehr.

URLs zu Projekten einbauen

5.1. Begriffe und Eigenschaften des relationalen Modells

Bei der Schaffung des Modells achtete man auf eine saubere Trennung der Schemas, d. h. konzeptionelles und internes Schema sollten getrennt sein. Die Darstellung der Daten erfolgt mittels Relationen.

Definition 5.1 (Relation)
Seien $D_1, D_2, \ldots, D_n$ Wertebereiche (Domains). Dann heißt $R \subseteq D_1 \times D_2 \times \ldots \times D_n$ mit $n \geq 1$ eine **Relation**.

Definition 5.2 (Grad, Stelligkeit)
Die Zahl n aus der Definition 5.1 wird als **Grad** oder **Stelligkeit** der Relation bezeichnet.

Definition 5.3 (Tupel, Komponente)
Ein Element $r \in R$ mit $r = (d_1, \ldots, d_n)$ für $d_i \in D_i$ heißt **Tupel** und d_i ist die ***i*-te Komponente** des Tupels.

Beispiel 5.1
Seien $D_1 = \{r, g, b\}$ und $D_2 = \{0,1\}$. Dann erhalten wir das kartesische Produkt $D_1 \times D_2 = \{(r,0), (r,1), (g,0), (g,1), (b,0), (b,1)\}$. Etwaige Relationen können sein: $R_1 = \{\}, R_2 = \{(r,0)\}, R_3 = \{(g,1), (b,0), (r,1)\}$.

Eventuell ist es besser, die Tabellen nebeneinander zu setzen.

Die Relationen können ebenfalls als Tabellen aufgefasst werden.

Tabelle 5.1.: Relation R_1

r	0

Tabelle 5.2.: Relation R_2

g	1
b	0
r	1

Tabelle 5.3.: Relation R_3

In der Relation R_3 gibt es die Tupel $(g,1), (b,0)$ und $(r,1)$.

Bisher wurden Relationen als Mengen von Tupeln betrachtet. Aus der Sicht der Datenbank ist ebenso das zugehörige Schema von Interesse, welchen den Relationstyp beschreibt.

Definition 5.4 (Relationsschema)
Ein **Relationsschema** besteht aus:

- Name des Schemas
- Menge von Attributnamen
- eventuellen zusätzlichen Integritätsbedingungen

Beispiel 5.2
Seien D_1 und D_2 die Domänen wie oben. Dann sind die Farbe und der Wert jeweils der **Attributname**. Die Farbtabelle ist der **Schemaname** und eine Integritätsbedingung könnte sein, dass die Farbe r nicht mit dem Wert 0 auftreten darf.

Beispiel 5.3
Wir betrachten das Relationsschema `ANGEST` mit der Attributmenge (Name, Beruf, Wohnort, Geburtsjahr). Dann ergibt sich folgende Tabelle:

Schemaname	Attributname			
Angest	Name	Beruf	Wohnort	Geburtsjahr
	Meier	Schlosser	Stuttgart	1967
	Müller	Schmied	Berlin	1977
	Petermann	Verkäuferin	Salzwedel	1987

Tabelle 5.4.: Relationsschema Angest

Die Menge von Relationsschemata und von zusätzlichen Integritätsbedingungen ergibt das **Datenbankschema**. Begrifflich wird zwischen Relationsschema und Relation nicht immer strikt unterschieden.

Wie bereits gesehen, sind Relationen Mengen. Das heißt, in einer Relation dürfen keine identischen Tupel auftauchen. Im Beispiel 5.3 wären zwei gleiche Zeilen der Form (Meier, Schlosser, Stuttgart, 1967) unzulässig. Weiterhin verlangt das Relationenmodell das Vorhandenseins eines Schlüssels nach Definition 2.5. Bei mehreren Kandidaten wird ebenso ein **Primärschlüssel** gewählt. Eine neue Form eines Schlüssels ist der Fremdschlüssel.

Definition 5.5 (Fremdschlüssel)
Falls eine Attributkombination einer Relation R_1 in einer Relation R_2 die Eigenschaft eines Primärschlüssels besitzt, so kann die in R_1 als Schlüssel gewählt werden. Dieser Schlüssel wird dann **Fremdschlüssel** bezeichnet.

Eine Konsequenz aus der Definition des Fremdschlüssels ist die **referentielle Integrität**. Dies ist neben der Integrität auf Datensatz- und Datenfeldebene eine Form der Datenintegrität.

Bemerkung 5.1
Eine Relation *muss* einen Primärschlüssel besitzen und *kann* beliebige Fremdschlüssel haben.

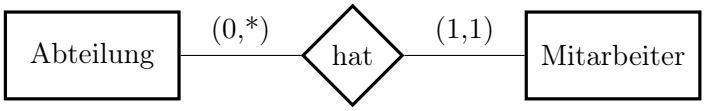
5.2. Abbildungen von E/R-Modell auf relationales Modell

5.2.1. Nichtrekursive 1:n-Beziehungstypen

Wir wollen den Entitytypen in ein Relationsschema überführen und betrachten das am Beispiel einer Abteilung mit zugeordneten Mitarbeitern. Die Abteilungsnummer bei Mitarbeiter ist der Fremdschlüssel in Bezug auf den Primärschlüssel von Abteilung. Es repräsentiert den Beziehungstyp „hat“ aus dem E/R-Diagramm. Die (atomaren) Attribute eines Entitytyps werden unmittelbar in Attribute der entsprechenden Relation

übernommen. Es ist zu beachten, dass beliebige Kardinalitäten, wie (3,7) nach der (min,max)-Notation, nicht darstellbar sind.

Das unten dargestellte E/R-Diagramm lässt sich in die Relation Abteilung(AbtNr, Abteilungsort etc.) und Mitarbeiter(PersNr, Name, AbtNr) überführen.



5.2.2. Rekursive 1:n-Beziehungstypen

noch verbessern, zwei Linien

Man stelle sich das folgende E/R-Diagramm vor:

Der Mitarbeiter hat (0,1) Vorgesetzte und der Vorgesetzte hat (0,*) Mitarbeiter. Das lässt sich wie folgt modellieren: Mitarbeiter(Personalnummer, Name, ..., Chefnummer). Die Chefnummer ist Fremdschlüssel mit Bezug auf den Primärschlüssel der gleichen Relation. Dann verbleibt nur noch die Frage, wie man in dem Beispiel den obersten Chef modelliert.

- Chefnummer und Personalnummer haben den gleichen Wert. Dies ist nicht empfehlenswert.
- Chefnummer ist undefiniert, d. h. es besitzt einen speziellen NULL-Wert.

5.2.3. Nichtrekursive n:m-Beziehungstypen

Dazu stellen wir uns vor, dass ein Lieferant Teile liefert. Die gelieferten Mengen können beliebig groß sein. Dies lässt sich in die Relation überführen:

- Lieferant(Liefernummer, Name, Ort)
- Teil(Teilenummer, Bezeichnung)

- Liefert(Liefernummer, Teilenummer, Menge)

Das heißt, der Beziehungstyp bekommt eine eigene Relation mit dem Primärschlüssel als der Attributkombination der Relationen Lieferant und Teil. Die Teilenummer in Liefert ist Fremdschlüssel mit Bezug auf Primärschlüssel von Teil und Liefernummer in Liefert ist Fremdschlüssel mit Bezug auf Primärschlüssel von Lieferant.

5.2.4. Rekursive n:m-Beziehungstypen

Dazu stelle man sich eine Stückliste mit den Beziehungen Teil enthält beliebige Unterteile und Unterteil hat beliebige Oberteile. Dies lässt sich mit zwei Relationen modellieren:

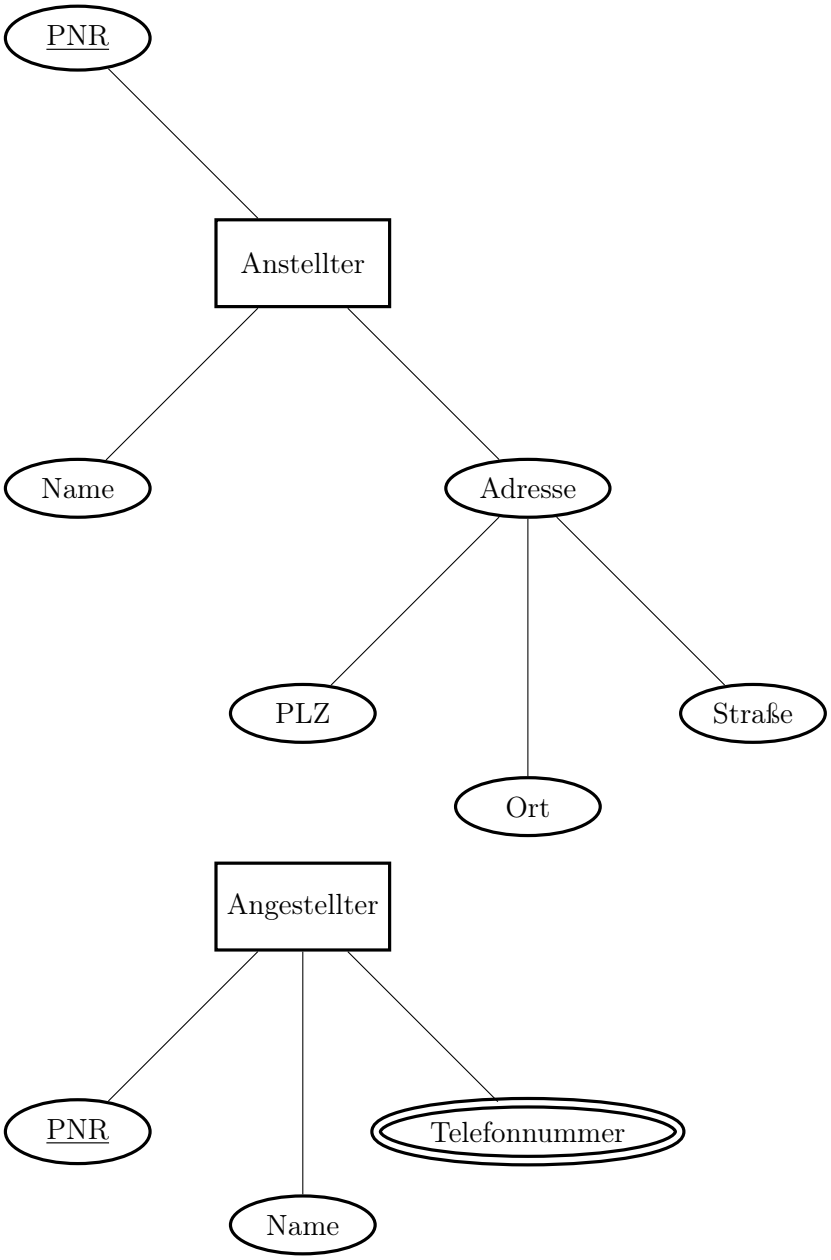
- Teil(Teilenummer, Bezeichnung)

- Struktur(Oberteilnummer, Unterteilnummer, Anzahl)

Es wird also wieder jeder Beziehungstyp in eine Relation überführt. Der Primärschlüssel setzt sich aus den Attributen Oberteilnummer und Unterteilnummer zusammen. Beide sind einzeln Fremdschlüssel mit Bezug auf den Primärschlüssel von Teil.

5.2.5. Umgang mit nichtatomaren Attributen

Zu Anfang stellen wir zwei Beispiele.



Das letzte Beispiel lässt sich mit zwei Tabellen lösen. Dabei ist die PNR Fremdschlüssel mit Bezug auf den Primärschlüssel von Angestellter. Das Problem der Nullwerte ist damit per Definition ausgeschlossen. Außerdem sind zusätzliche Integritätsbedingungen denkbar.

Will man die Daten in eine Tabelle modellieren, entstehen schnell Konflikte. Denn als Primärschlüssel empfiehlt sich eine Kombination aus PNR und Telefonnummer. Wenn ein Mitarbeiter keine Telefonnummer hat, so ist der Primärschlüssel nicht definiert.

Angestellter	PNR	Name
	3841	Schmid
	6513	Meier
	8612	Lehman

Tabelle 5.5.: Tabelle mit PNR und Name

Telefon	PNR	Telefonnummer
	3841	23813
	3841	54128
	6513	12908
	8612	87124
	8612	98236
	8612	76123

Tabelle 5.6.: Tabelle mit PNR und Telefonnummer

Das erste obige Beispiel mit Adressen könnte man in einer Tabelle modellieren. Dabei geht die Zusammengehörigkeit der Adressdaten (PLZ, Ort, Straße) verloren. Daher ist es auch hier zu empfehlen, zwei Tabellen zu verwenden.

5.3. Relationenalgebra und -kalkül

Die Sprachen für das relationale Modell sind nicht navigierend und satzorientiert, sondern mengenorientiert und deskriptiv. Mit einer Anweisung werden Mengen von Tupeln gelesen bzw. modifiziert. Zu den Basisoperationen des relationalen Modells zählen:

Selektion Auswahl von Tupeln mit einer Relation

Projektion Streichen von Spalten

Verbund (Join) Verknüpfung von Tabellen aufgrund von Attribut-Beziehungen

Mengenoperation Vereinigung, Differenz und Durchschnitt auf verschiedenen Relationen gleicher Struktur

Beispiel 5.4
Finde die Wohnorte aller Angestellten, die Programmierer sind. Dazu werden alle Tupel der Angestellten-Relation selektiert, deren Beruf Programmierer ist. Daraufhin folgt eine Projektion auf die Spalte Wohnort. Das Ergebnis der Operation ist wieder eine Menge.

Finde die Namen aller Angestellten, die am Projekt Foo mitarbeiten. Dazu werden die Relationen Angestellter und Mitarbeit miteinander verbunden. Dies geschieht, indem Tupel

mit gleicher Angestelltennummer zusammengefasst werden. Auf dem Zwischenergebnis erfolgt eine Selektion der Tupel mit Projektnummer 27 und eine Projektion auf die Spalte Name. Andere Lösungen sind auch denkbar.

5.3.1. Kriterien für Anfragesprachen

In dem Buch „Datenbanken kompakt“ von Heuer, Sattler und Saake (siehe [2]) sind auf Seite 225 verschiedene Kriterien festgelegt. Dazu zählen:

Ad-hoc-Formulierung Benutzer kann eine Anfrage ohne zugehöriges Programm formulieren.

Mengenorientiert Jede Operation arbeitet auf Mengen von Daten.

Deskriptivität Benutzer formuliert, was er haben will.

Abgeschlossenheit Ergebnis der Anfrage ist wieder eine Relation und kann als Eingabe verwendet werden.

Adäquatheit Alle Konstrukte des Datenmodells werden unterstützt.

Orthogonalität Sprachkonstrukte sind in ähnlichen Sprachen anwendbar. Orthogonalität setzt Abgeschlossenheit voraus.

Optimierbarkeit Sprache besteht aus wenigen Operationen, für die es Optimierungsregeln gibt. Die Eigenschaft wird durch eine formale Sprachdefinition sowie durch die Orthogonalität gefördert.

Effizienz Jede Operation ist effizient ausführbar.

Sicherheit Keine syntaktisch korrekte Anfrage darf in eine Endlosschleife führen oder ein unendliches Ergebnis liefern.

Eingeschränktheit Sprache darf keine vollständige Programmiersprache sein.

Vollständigkeit Sprache muss mindestens die Anfragen aus der Relationenalgebra ausdrücken können.

5.3.2. Relationenalgebra

Der Begriff der **Relationenalgebra** stammt vom Begriff der Algebra aus der Mathematik. Es ist eine nicht leere Menge mit Operationen, die auf dem Wertebereich definiert sind. Übertragen auf das relationale Modell heißt dass, dass die Werte die Relationen darstellen und die Operationen sind die vorgestellten Basisoperationen.

Beispiel 5.5
Im folgenden betrachten wir die zwei unten stehenden Beispieltabellen. Diese sind [2] entnommen.

Buch	Inventarnummer	Titel	ISBN	Autor
	7613	Dr. No	3145	James B
	8712	ObjektDB	6512	Heuer
	5412	Analysis I	9823	Bauer
	9131	COBOL	6192	Würth

Tabelle 5.7.: Relation Buch

Ausleihe	Inventarnummer	Name
	7613	Meier
	8712	Müller
	5412	Schulze
	9131	Meier

Tabelle 5.8.: Relation Ausleihe

Nun wollen wir verschiedene Operationen diskutieren.

Projektion Üblicherweise wird die Projektion durch $\pi_{\text{Attr}}R$ bezeichnet. Dabei steht Attr für eine nicht leere Menge von Attributen der Relation R. Die Projektion $\pi_{\text{Attr}}R$ wählt jene Spalten von R aus, die in Attr gegeben sind.

So besteht zum Beispiel $\pi_{\{\text{Name}\}}$Ausleihe aus Meier, Müller und Schulze. Wegen der Mengenoperation wurden die Duplikate entfernt. Wenn das Attribut einen Schlüssel enthält, ist hingegen keine Elimination möglich.

Selektion Die Selektion wird durch $\sigma_F R$ bezeichnet. Dabei steht F für eine Selektionsformel. Die Operation $\sigma_F R$ wählt alle Zeilen von R aus, die der Bedingung F genügen. Beispielsweise gibt $\sigma_{\text{Inventarnummer}>8800}$Buch die Zeile (9131, COBOL, 6192, Würth) zurück. Man kann auch Attributwerte untereinander vergleichen und logische Verknüpfungen sind ebenso möglich.

Verbund (Join) Der Verbund wird durch $R \bowtie_F S$ bezeichnet. Dabei steht F für eine Verbundbedingung. Der Verbund $R \bowtie_F S$ fügt die Tupel der Relationen R und S zusammen, die der Bedingung F genügen.

Mittels Buch $\bowtie_{\text{Inventarnummer}=\text{Inventarnummer}}$ Ausleihe fügt man beide Tabellen ineinander. Die Gleichheitsbedingung wird als **Equi-Join** bezeichnet.

Vereinigung Die Vereinigung $R \cup S$ ist bei gleichen Relationsschemata möglich. Bei unterschiedlichen Relationsschemata ist zu entscheiden, ob die Anzahl der Attribute (Spaltenzahl) und der Wertebereich kompatibel ist. In dem Fall kann die Voraussetzung für eine Vereinigung geschaffen werden.

Durchschnitt Die Voraussetzungen für den Durchschnitt entsprechen denen der Vereinigung. Wir schreiben $R \cap S$.

Differenz Die Differenz $R \setminus S$ enthält alle Tupel von R, die nicht in S vorkommen. Die Voraussetzung für die Anwendung entspricht der der Vereinigung.

Es lassen sich weitere Basisoperationen aufzählen. Jedoch bilden die oben aufgeführten eine minimale Relationenalgebra. Das bedeutet, dass das Weglassen einer Operation die Mächtigkeit reduziert. Hingegen bringt ein Hinzufügen einer neuen Operation keine Steigerung der Mächtigkeit.

Operationen lassen sich in äquivalente Operationen umformen. Das ist jedoch nicht beliebig möglich. Es darf nur eine syntaktische Transformation ohne Änderung der Semantik geben. Eine Auswahl äquivalenter Umformungen ist:

1. $\sigma_{F_1}(\sigma_{F_2}R) \equiv \sigma_{F_2}(\sigma_{F_1}R)$
2. $\sigma_F R \equiv \sigma_{F_1}(\sigma_{F_2}R)$ mit $F = F_1 \wedge F_2$
3. $\sigma_F(\pi_{\text{Attr}}R) \equiv \pi_{\text{Attr}}(\sigma_F R)$, wenn die Attribute von F eine Teilmenge von Attr sind.
4. $S \cup R \equiv R \cup S$
5. $S \bowtie_F R \equiv R \bowtie_F S$
6. $(R \cup S) \cup T \equiv R \cup (S \cup T)$

5.3.3. Relationenkalkül

Das Relationenkalkül besteht aus zwei Komponenten, dem Tupel- und dem Domänenkalkül. Beide sind recht ähnlich und können daher zusammen betrachtet werden. Grundsätzlich wird dabei beschrieben, welche Bedingungen (Prädikate) die Tupel der Ergebnisrelation erfüllen müssen.

Definition 5.6 (Tupelvariable)
Eine **Tupelvariable** ist ein Tupel einer Relation. Sei die Tupelvariable U der Relation R zugeordnet und A ein Attribut von R. Dann bezeichnet $U.A$ den Attributwert von A in irgendeinem Tupel von R. Dies heißt auch **Tupelkomponente**. Sind x, y Konstanten oder Tupelkomponenten, so legt $x\theta y$ mit $\theta \in \{=, \neq, <, \leq, >, \geq\}$ eine **Bedingung** fest.

Definition 5.7 (Formel)
Eine **Formel** wird durch folgende Konstruktionsvorschriften definiert:

1. Jede Bedingung ist eine Formel.
2. Ist f eine Formel, so auch (f) und $\neg(f)$.
3. Sind f und g Formeln, so auch $f \wedge g$ und $f \vee g$.
4. Ist f eine Formel und T eine Tupelvariable in f, so sind auch $\exists T(f)$ und $\forall T(f)$ Formeln.
5. Nur durch die obigen Vorschriften erzeugbaren Ausdrücke sind Formeln.

5.4. Structured Query Language

Die Structured Query Language (SQL) ist eine Datenbanksprache zur Definition, Abfrage und Manipulation von Daten. Es ist als Norm der Reihe DIN ISO/IEC 9075 festgelegt. Die Bezeichnung der Sprache stammt vom Vorgänger SEQUEL (für Structured English Query Language) ab. Das Wort war eingetragenes Warenzeichen einer anderen Firma. Daher erfolgte eine Umbenennung. Seit 1982 ist die Sprache genormt. Seither gab es folgende Schritte:

- 1986 wird SQL1 als ANSI-Standard verabschiedet
- 1987 wird SQL1 von der ISO als Standard verabschiedet
- 1992 wird SQL2 oder SQL-92 von der ISO als Standard verabschiedet
- 1999 SQL3 oder SQL:1999
- 2003 Standard SQL:2003 ISO/IEC 9075:2003 als Nachfolger von SQL3
- 2006 Standard SQL:2006 ISO/IEC 9075-14:2006 Standardisierung im Zusammenhang mit XML
- 2008 aktuelle Revision SQL:2008 ISO/IEC 9075:2008

5.4.1. Datendefinition mit SQL

Innerhalb von SQL sind Datendefinition und -manipulation vereinheitlicht. Einige Anweisungen getrennt nach der jeweiligen Ebene finden sich in Tabelle 5.9.

5.4.2. SQL-Anweisungen

Im folgenden sollen einige SQL-Anweisungen beispielhaft erwähnt und erklärt werden.

Ebenen		
externe	konzeptionelle	interne
create view	**create table**	**create index**
drop view	**alter table**	**alter index**
	drop table	**drop index**
	create domain	
	alter domain	
	drop domain	

Tabelle 5.9.: Überblick zu Datendefinitionsanweisungen

CREATE TABLE

Die Anweisung **CREATE TABLE** erzeugt eine k-spaltige Tabelle. Gemäß unten stehender Syntax wird das Relationenschema über spalteK/wertebereichK definiert.

```
CREATE TABLE relationenschema
(spalte1 wertebereich1 [NOT NULL]
\ldots{}
spalteK wertebereichK [NOT NULL])
```

Als Datentypen existieren in der Regel Integer, Float, Character und andere produktspezifische. Die obige Klausel **NOT NULL** verbietet das Auftreten von nicht definierten Werten.

Der aufmerksame Leser wird festgestellt haben, dass kein Schlüssel zugewiesen wurde. Die SQL-Norm erzwingt keinen Primärschlüssel. Daher können Tabellen durchaus Duplikate aufweisen. Mit der Anweisung **PRIMARY KEY** (spalteN) wird ein Primärschlüssel zugewiesen und mittels **FOREIGN KEY** (spalteN) ein Fremdschlüssel.

Die Klausel **default** erlaubt es einen Standardwert anzugeben und mit **check** werden Integritätsbedingungen angegeben.

Nach dem Anlegen der Tabelle ist diese zunächst leer. Die Schemainformationen sind in einem **Datenbankkatalog** abgelegt. Dieser wird manchmal als **Data Dictionary** bezeichnet und enthält alle mit **CREATE TABLE** erzeugten Tabellen, alle Attribute, benutzerdefinierte Wertebereiche etc. Katalogtabellen können wir „normale“ Tabellen mittels SQL gelesen werden. Dies ist ein wesentlicher Vorteil gegenüber den anderen Modellen. Dort waren diese Kataloginformation in einer internen Form gespeichert.

Die Definition von Primär-/Fremdschlüssel zielt auf die Sicherung der Integrität ab. Die Integritätssicherung war eines der großen Themen bei der Normung. Folgende Zusätze wurden geschaffen:

`default`-Klausel dient zur expliziten Festlegung eines Standardwertes für eine Tabellenspalte (siehe Listing 5.1).

`check`-Klausel dient zur Spezifikation von Integritätsbedingungen einzelner Attributwerte. In Listing 5.2 erfolgt eine Prüfung, dass der Primärschlüssel zwischen 5 und 999 liegt sowie dass der Autor nicht den Namen „Kujau“ trägt. Die `check`-Klausel ermöglicht die Angabe von Prädikaten, die den zulässigen Wertebereich von Attributen weiter einschränken. Die Kontrolle geschieht durch das DBMS beim Einfügen oder Ändern der Werte. Bei einer drohenden Verletzung der Regeln wird die Anweisung *nicht* ausgeführt, sondern zurückgewiesen. Daher sind Anweisungen stets **atomar**.

Listing 5.1: CREATE TABLE mit default-Wert

```
CREATE TABLE Buch
   (InvNr integer primary key,
    Titel varchar(30),
    ISBN char(5),
    Autor varchar(40) default   'Kuespert')
```

Listing 5.2: CREATE TABLE mit check-Wert

```
create table Buch
   (InvNr integer primary key
        check (InvNr between 5 and 9999),
    Titel varchar(30),
    ISBN char(5),
    Autor varchar(40) default   'Kuespert'
        check (Autor != 'Kujau'))
```

CREATE DOMAIN

Diese Anweisung ermöglicht die Vereinbarung benutzerspezifischer Wertebereiche. Die oben vorgestellten `default`- und `check`-Klauseln dürfen mit verwendet werden. Ein Beispiel ist die Angabe eines Fachgebiets:

```
CREATE DOMAIN Fachgebiet varchar (20)
  default 'Informatik'
CREATE TABLE buch
(invnr integer PRIMARY KEY,
Gebiet Fachgebiet,
titel varchar (30),
ISBN char (13),
Autor varchar (30))
```

ALTER TABLE

Die Anweisung dient zum Hinzufügen neuer Attribute zu einer Tabelle bzw. zum Löschen von Tabellenspalten. Falls es Probleme bei der Anwendung gibt, ist zu beachten, dass die obigen Klauseln erst Bestandteil von SQL2 sind. Bereits in der älteren Definition ist die Anweisung **ALTER TABLE** basisrelation **ADD** spalte wertebereich enthalten. Sie fügt eine neue Spalte ein. Bei allen vorhandenen Tupeln wird sie mit **NULL** besetzt. Mittels **ALTER TABLE** basisrelation **DROP** spalte lässt sich eine Spalte löschen.

Die meisten Datenbanksysteme unterstützen nur **ALTER TABLE** basisrelation **ADD**. Insbesondere die Änderung durch **ALTER TABLE** basisrelation **ALTER** wird in den Systemen nicht umgesetzt. Der Hauptgrund ist das heikle Thema einer Änderung der Definition. Was passiert mit den vorhandenen Daten? Sollen diese sofort oder erst verzögert in das neue Format transformiert werden? Was ist, wenn die Transformation scheitert? Diese Fragen sind zum Teil nur im Einzelfall zu behandeln.

DROP TABLE

Mit der Anweisung **DROP TABLE** basisrelation wird eine Tabelle, d. h. Relationenschema und Daten, gelöscht. Durch das Anfügen von **restrict** wird das Löschen verhindert, wenn noch auf die Daten Bezug genommen wird. Mit **cascade** werden bezugnehmende Integritätsbedingungen und Sichten mit gelöscht.

```
DROP TABLE buch cascade;
DROP TABLE buch restrict;
```

CREATE/DROP INDEX

Die nächste Anweisung erzeugt oder löscht einen **Zugriffspfad** (**Index**). Dieser erlaubt einen schnellen Zugriff auf bestimmte Tupel bei einem gegebenen Attributwert.

```
CREATE [UNIQUE] INDEX indexname
  ON basisrelation
  (spalte1 ordnung1,
    ...
  spalteN, ordnungN)
```

Die Angabe ordnungI steht dabei für **ASC** bzw. **DESC**. Falls **UNIQUE** verwendet wird, so müssen die Werte im Index eindeutig sein. Wenn bei **CREATE TABLE** ein **PRIMARY KEY** vorgegeben wurde, wird für diesen ein eindeutiger Index erzeugt. Dies ist zwar keine Forderung der SQL-Norm, aber Realität in den Produkten.

UPDATE

Nun folgen einige Anweisungen zur Änderung von Daten innerhalb der Tabellen. Mittels **UPDATE** werden bestehende Daten überschrieben. Die Syntax ist wie folgt:

```
UPDATE basisrelation
  set spalte1 = ausdruck1
  ..
  set spalteN = ausdruckN
  [WHERE bedingung]
```

Die Attributwerte werden in allen Spalten, die die angegebene Bedingung erfüllen, ersetzt.

Der Roman „Verdammnis“ von Stieg Larsson wurde in der Datenbank fehlerhaft erfasst. Daher soll der Eintrag verbessert werden. Die Inventarnummer des Buches ist 8.

```
UPDATE buch
  set titel = 'Verdammnis'
  WHERE InvNr = 8
```

Einige Bibliotheksbestände werden ausgelagert. Das betrifft alle Titel mit der Inventarnummer größer als 7500.

```
UPDATE buch
  set teilbibo = 'Leutragraben 5'
  WHERE InvNr > 7500
```

Der Primärschlüssel InvNr soll für alle Bücher auf 9999 gesetzt werden.

```
UPDATE buch
  set InvNr = 9999
```

Diese Operation schlägt fehl. Denn die Grundvoraussetzung für den Primärschlüssel wird verletzt. Im Allgemeinen lassen sich einzelne Inventarnummern ändern, sofern der neue Wert noch nicht belegt ist.

DELETE

Alle Tupel einer Basisrelation, die eine bestimmte Bedingung erfüllen, sollen gelöscht werden.

```
DELETE FROM basisrelation
  WHERE bedingung
```

Unter Umständen kann eine Löschung zu einer Verletzung der Integritätsbedingungen führen.

INSERT

Nun wollen wir neue Werte einfügen. Dazu dient **INSERT**. Diese Anweisung kann in zwei Erscheinungsformen auftreten. Zum einen können einfach Tupel als Konstanten eingefügt werden. Andererseits können berechnete Werte aus anderen Relationen eingefügt werden. Die unten dargestellte Syntax ist vereinfacht:

```
INSERT INTO basisrelation [(spalte1,..,spalteN)]
  VALUES (konstante1,..,konstanteN)
```

So können wir ein neues Tupel in die Relation buch einfügen:

```
INSERT INTO buch (InvNr, ISBN)
  VALUES (9382, 9-882-23-1235-4)
```

Was passiert mit den Werten von Autor und Titel? Diese werden auf **NULL** gesetzt. Werte, die bei der Anweisung **CREATE TABLE** mit **NOT NULL** festgelegt werden, müssen auf jeden Fall belegt werden. Ansonsten schlägt das Einfügen fehl.

Weiterhin kann eine SQL-Anfrage die einzufügenden Werte vorgeben. Beispielsweise könnte es eine Tabelle geben, in der Bücher aufgeführt sind, die seit längerem nicht ausgeliehen wurden. Diese sollen nun in die Originaltabelle eingefügt werden. Wie sieht die SQL-Anweisung aus?

```
INSERT INTO Ausleihe
(SELECT * FROM Ausleihe-ganz-alt)
```

SELECT FROM

Die oben vorgestellten Aktionen waren allesamt Schreibzugriffe auf die Datenbank. Nun soll es um Lesezugriffe gehen. Fundamental ist das so genannte **SELECT FROM WHERE**-Konstrukt (**SFW-Konstrukt**). Diese Konstrukt kann um weitere Klauseln erweitert werden. Die Klausel **SELECT** legt die Ergebnisstruktur (Projektionsliste) einer Abfrage fest. Woher das DBMS die Daten holen soll, erfährt es durch die Angabe im **FROM**. Das heißt, welche Relationen werden zur Anfrage benötigt. Schließlich legt die **WHERE**-Klausel die Selektionsbedingung und eventuell eine Datenverknüpfung fest. Seit SQL:1992 gibt es hierfür einen eigenen **JOIN**-Operator, wie wir bereits in der Relationenalgebra (Abschnitt 5.3.2) kennen gelernt haben.

Syntax der Anweisung als Grafik einbinden

Es sei unsere Angestellten-Projekt-Datenbank gegeben. Gesucht sind alle Angestellten mit Name und Wohnort, die in einer Abteilung mit einer Nummer größer als 3 arbeiten. Die Ausgabe soll nach Name absteigend sortiert werden und alle Duplikate sind zu erhalten. Die Anfrage könnte folgendermaßen aussehen:

```
SELECT ALL Name, Wohnort
FROM Angest
WHERE AbtNr > 3
ORDER BY Name DESC
```

Dabei ist **ALL** ein Standardwert und kann weggelassen werden. Das Ergebnis der Abfrage liegt streng genommen nicht mehr im relationen Modell. Denn der Mengencharakter wird durch die Duplikate verletzt.

Nun wollen wir eine Selektion in SQL darstellen. Dazu soll die Anfrage alle Angestellten ausgeben, die von Beruf „Hundezüchter“ sind.

```
SELECT *
FROM Angest
WHERE Beruf = 'üHundezchter'
```

Der Stern in der Anweisung sorgt dafür, dass alle Attribute ausgegeben werden. Alternativ liessen sich alle einzeln angeben.

Wie sieht ein Projektion aus? Wir wollen für alle Angestellten den Beruf und den Wohnort wissen.

```
SELECT Beruf,Wohnort
FROM Angest
WHERE true
```

Die **WHERE**-Klausel wird üblichweise weggelassen und ist unnötig, da Informationen zu *allen* Angestellten abgefragt werden. Die Reihenfolge der Attribute in der Ergebnisrelation ist eigentlich bedeutungslos. Jedoch kann sie in der Ausgabe aus Gründen der Lesbarkeit wesentlich sein. Die Angabe von **DISTINCT** würde dafür sorgen, dass Duplikate eliminiert werden.

Im nächsten Schritt wollen wir Projektion und Selektion kombinieren. Die Anfrage soll die Namen aller Angestellten ausgeben, deren Abteilungsnummer zwischen 2 und 6 liegt.

```
SELECT Name
FROM Angest
WHERE AbtNr >= 2 AND AbtNr <= 6
% oder
WHERE AbtNr BETWEEN 2 AND 6
% oder
WHERE AbtNr IN (2,3,4,5,6)
```

Nach **IN** folgt eine Multimenge atomarer Werte. Diese darf auch leer oder einelementig sein. Typischerweise wird sie als Ersatz für die **OR**-Verknüpfung benutzt. So kann man statt AbtNr=12 **OR** AbtNr=23 **OR** AbtNr=27 besser AbtNr **IN** (12,23,27) schreiben.

Nach **IN** darf ein SFW-Ausdruck stehen. Dieser wird **Subquery** oder **Subselect** genannt und Gegenstand späterer Betrachtungen sein.

Nun ist es recht nützlich, sortierte Ergebnisse einer Anfrage zu erhalten. Beispielsweise möchten wir die Attribute Name und Wohnort aus der Relation Angest ausgegeben haben. Dabei soll die Ausgabe nach dem Namen sortiert sein.

```
SELECT Name, Wohnort
FROM Angest
ORDER BY Name ASC
```

Die Angabe von **ASC** kann auch weggelassen werden, da dies der Standardfall ist. Absteigend können die Werte mittels **DESC** sortiert werden. Nun stelle man sich vor, es gibt mehrere gleiche Namen und unterschiedliche Wohnorte. Da liegt die Frage nahe, wie wohl die Wohnorte sortiert werden. Ohne besondere Anweisungen werden diese unsortiert („zufällig“) ausgegeben. Wenn jedoch mehrere Angaben in der **ORDER BY**-Klausel stehen, legt dies eine Priorität fest. Das heißt, wenn eine Sortierung nach Wohnort gewünscht ist, könnte der Nutzer dies als zweiten Wert angeben. Statt des Attributnamens kann eine Attributnummer, also die Position des Attributs, verwendet werden. Das bietet sich insbesondere an, wenn der Name lang ist oder es sich um ein berechnetes Feld handelt. Gegebenenfalls gibt es hierfür besser geeignete Methoden.

Neben den oben vorgestellten Funktionen gibt es weitere eingebaute Funktionen. Unter anderem handelt es sich dabei um **COUNT**, **SUM**, **AVG**, **MIN**, **MAX**. So kann die Fragen nach der aktuellen Anzahl der Angestellten der Firma einfach beantwortet werden:

```
SELECT COUNT(*)
FROM Angest
```

Recht nützlich erweisen sich die eingebauten Funktionen in Zusammenhang mit der **ORDER BY**-Klausel. Eine Liste aller Wohnorte mit dem durchscnittlichen Resturlaub der Angestellten, die in dem Ort wohnen, erzeugt man mit:

```
SELECT Wohnort, AVG(Resturlaub)
FROM Angest
ORDER BY Wohnort
```

Das DBMS wendet die Bildung des Durchschnitts nicht auf die komplette Liste an, sondern gruppiert sie in gleiche Wohnorte. Gerade beim Durchschnitt ist es unter Umständen sinnvoll, Werte zu betrachen, die eine bestimmte Mindestanzahl besitzen. Im folgenden Beispiel werden mindestens drei Einträge verlangt.

```
SELECT Wohnort, AVG(Resturlaub)
FROM Angest
ORDER BY Wohnort
HAVING COUNT (*) >= 3
```

Nun wollen wir gern den oder die Angestellten wissen, die den geringsten Resturlaub besitzen. Die einfachste Variante wäre eine nach dem Resturlaub sortierte Liste auszugeben. Bei einer großen Zahl Angestellten kann die Ausgabe entsprechend lange dauern. Somit ist es sinnvoll, mittels der **WHERE**-Klausel die Anfrage einzuschränken. Hier kommen die oben angesprochenen Subquerys ins Spiel. Wie der Name vermuten lässt, sind diese ebenfalls Abfragen, die innerhalb einer Abfrage ausgeführt werden. Das Ergebnis wird in der Originalabfrage weiter verwendet. Die korrekte Anfrage könnte so aussehen:

```
SELECT *
FROM Angest
WHERE Resturlaub = (SELET MIN (Resturlaub) FROM Angest)
```

Innerhalb einer neue Anfrage, wollen wir alle Angestellten wissen, die in Hamburg, Heidecksburg oder Heinzburg wohnen. Im Rahmen der obigen Darstellungen haben wir bereits **OR** kennengelernt und könnten dies hier anwenden. Eine weitere Möglichkeit für eine „unscharfe“ Suche sind Wildcards in Verbidung mit dem Vergleichsoperator **LIKE**. Der Prozentzeichen (%) umfasst beliebig viele Zeichen und der Unterstrich (_) umfasst genau ein Zeichen. Somit lässt sich die obige Anfrage wie folgt festlegen:

```
SELECT *
FROM Angest
WHERE Wohnort LIKE 'H%burg'
```

Der Vergleichsoperator **LIKE** darf negiert werden und kennt keine Wortsemantik. Das heißt, Wortgrenzen sind dem Operator nicht bekannt.

Das Kreuzprodukt der Tabellen Angest und Projekt lässt sich über **SELECT** * **FROM** Angest, Projek ausgeben. Das DBMS erzeugt ein Ergebnis, welches alle Spalten der beteiligten Relationen enthält und jedes Tupel der einen Relation mit jedem Tupel der anderen Relation verknüpft. In der Regel wird man das Kreuzprodukt kaum benötigen. Stattdessen werden Bedingungen angegeben, unter der die Tupel verknüpft werden. Bespielsweise kann für jeden Angestellten ausgegeben werden, in welchen Projekt er zu wieviel Prozent arbeitet:

```
SELECT a.name, m.pronr, m.prozent
FROM Angest a, Mitarb m
WHERE a.AngNr = m.AngNr
```

Nun wollen die obige Ausgabe für Mitarbeiter aus Erfurt und für Projekte, deren Beschreibung nicht „Wartburg“ enthält

```
SELECT .. FROM ..
WHERE a.Wohnort = 'Erfurt' AND p.Beschr NOT LIKE '%Wartburg'
```

Der Inner Join ist die am häufigstens verwendete Join-Operation und muss daher in SQL nicht explizit angegeben werden. Damit erhält der Abfragende nur Werte für die es in

den jeweiligen Relationen ein „Gegenstück“ gibt. Ist dies nicht der Fall, so erfolgt keine Ausgabe. Beispielsweise lassen sich die Mitarbeiter, die in keinem Projekt mitarbeiten bzw. Projekte ohne Mitarbeiter nicht mittels einem Inner Join ausgeben. Seit SQL92 gibt es hierfür eine Lösung, den Outer Join. Dieser lässt sich in die drei Teile aus Abbildung 5.1 zerlegen.

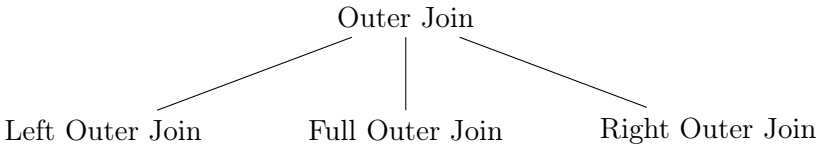


Abbildung 5.1.: Verschiedene Ausprägungen des Outer Join

Der Left Outer Join sei hier an den Relationen Angest und Mitarbeit durchgeführt. Wenn der Mitarbeiter mit der AngNr 205 in keinem Projekt tätig ist, so erscheint er trotzdem im Ergebnis der Abfrage. Die undefinierten Werte für ProNr und Prozent sind mit **NULL** belegt. Laut der Syntax von SQL92 sieht die Abfrage wie folgt aus:

```
SELECT a.AngNr, a.Name, a.Wohnort, a.Beruf, a.AbtNr,
  m.ProNr, m.Prozent
FROM (Angest a LEFT OUTER JOIN Mitarbeit m ON a.AngNr = m.AngNr)
```

Für den Right Outer Join nehmen wir an, dass das Projekt mit der ProNr 94 neu initiiert wurde und noch über keine Mitarbeiter verfügt. Im Ergebnis der Abfrage wird dann das Feld AngNr mit **NULL** belegt. Die Abfrage gestaltet sich folgendermaßen:

```
SELECT m.AngNr, m.ProNr, p.PName
FROM (Mitarbeit m RIGHT OUTER JOIN Projekt p ON m.ProNr = p.ProNr)
```

Für den Full Outer Join finden sich in unseren Beispieltabellen keine Anwendungen. In dem Falle bleiben die Tupel sowohl aus dem linken wie auch aus dem rechten Operanden bestehen. Weitere Joins in SQL92 sind **CROSS JOIN**, **UNION JOIN** und **NATURAL JOIN**. Das Ergebnis der Abfragen kann auch anders modelliert werden. Jedoch erspart die explizite Notation in der Regel Schreibarbeit und vermeidet Fehler.

Subquerys

Eine **Subquery** ist eine Anfrage, in dessen **WHERE**-Klausel wieder ein SFW-Konstrukt auftritt. Dies kann beliebig tief geschachtelt werden. Die Syntax kann in drei Ausprägungen auftreten:

1. **SELECT** .. **FROM** .. **WHERE** [**NOT**] **EXISTS** (Subquery)

2. **SELECT** .. **FROM** .. **WHERE** op {**ANY**,**ALL**} (Subquery); dabei steht op für eines der Zeichen aus $\{<, \leq, =\neq, \geq, >\}$.
3. **SELECT** .. **FROM** .. **WHERE IN** (Subquery)

Man kann zwischen **korrelierten** und **unkorrelierten Subquerys** unterscheiden. Bei einer korrelierten Subquery nimmt die innere Abfrage Bezug auf die äußere. Ein Beispiel lässt sich über die Ausgabe aller Nummern und Namen von Angestellten konstruieren, die in einem Projekt zu 100 % mitarbeiten:

```
SELECT a.AngNr, a.Name
FROM Angest a
WHERE EXISTS
   (SELECT *
    FROM Mitarbeit m
    WHERE a.AngNr = m.AngNr AND m.Prozent = 100)
```

In dem Fall nimmt das a Bezug zur außen. Durch **EXISTS** wird getestet, ob das Ergebnis der Subquery nicht leer ist. Selbiges Ergebnis liesse sich ebenfalls durch eine unkorrelierte Subquery erreichen:

```
SELECT a.AngNr, a.Name
FROM Angest a
WHERE a.AngNr = ANY
   (SELECT AngNr
    FROM Mitarbeit
    WHERE Prozent = 100)
```

Hier wird innerhalb der Subquery kein Bezug zu außen genommen und im Gegensatz zu oben wird die innere Anfrage nur einmal durchlaufen. Mit diesem Zwischenergebnis wird wegen der Angabe von **ANY** geprüft, ob *irgendein* Wert der Subquery mit dem außen angegebenen Wert übereinstimmt. Würde dort **ALL** stehen, so müsste der außen angegebene Wert mit *allen* Werten aus dem Zwischenergebnis übereinstimmen. Sollte keines der beiden verwendet worden sein, so muss die Subquery genau ein Tupel zurückliefern.

Sichten in SQL

Mit Sichten lassen sich virtuelle Relationen erzeugen. Diese gehen aus Basisrelationen oder aus bereits erzeugten Sichten hervor. Dabei ist eine Basisrelation eine solche, die mittels **CREATE TABLE** erzeugt werden kann. Dieses Sprachmittel wurd geschaffen, um die externe Ebene gemäß der ANSI/SPARC-Architektur (siehe Abschnitt 1.4.1) nachzukommen. Der Benutzer der Datenbank ist somit entkoppelt von den tiefer liegenden Ebenen. Mit Sichten lassen sich für verschiedene Nutzergruppen zugeschnittene Ergebnisse produzieren. In Abbildung 5.2 ist das Syntaxdiagramm dargestellt.

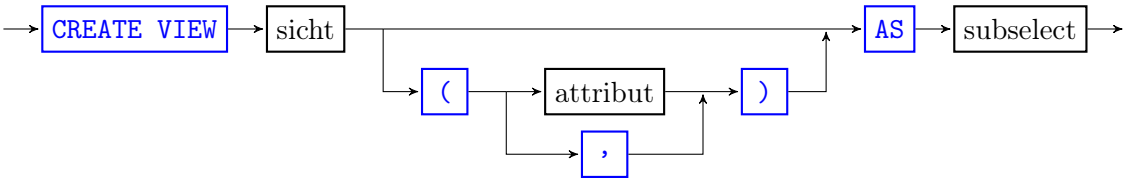


Abbildung 5.2.: Syntaxdiagramm zur Definition von Sichten in SQL

Weitere Anweisungen einfügen. Die weiteren SQL-Anweisungen werden eventuell später nachgetragen. Details bitte vorerst der Dokumentation entnehmen

Teil II.

Datenbanksysteme 2

6. Transaktionsverwaltung und Fehlerbehandlung

6.1. Transaktionen

Definition 6.1 (Transaktion)
Eine **Transaktion** ist eine Folge zusammengehöriger Operationen auf der Datenbank für die bestimmte Eigenschaften gelten.[1]

Bemerkung 6.1
Bei einer Transaktion kann es sich um Lese- oder Schreiboperationen handeln. Man kann sich das als Klammerung von Operationen vorstellen. Beginn und Ende der Transaktion werden entweder vom Benutzer oder von der Anwendung markiert. Operationen finden auf Datenbanken ausschließlich im Rahmen von Transaktionen statt. Im Extremfal ist jede Operation eine Transaktion.

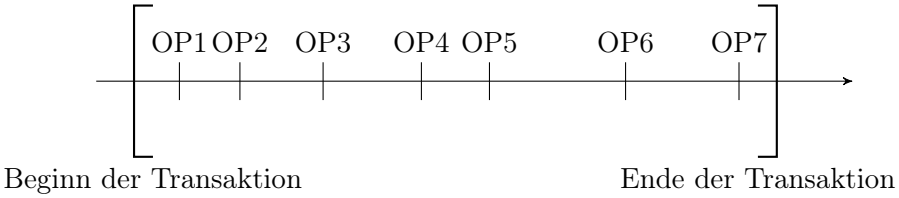


Abbildung 6.1.: Folge von Operationen innerhalb einer Transaktion

Definition 6.2 (Eigenschaften von Transaktionen)
Atomarität Eine Transaktion wird entweder komplett oder gar nicht durchgeführt.

Konsistenz Eine Transaktion überführt die Datenbank von einem logisch konsisten Zustand in einen logisch konsisten Zustand, d. h. alle Integritätsbedingungen gelten.

Isolation Die Operationen innerhalb einer Transaktion werden erst nach Abschluss der Transaktion nach außen sichtbar.

[1] Jim Gray gilt als der Erfinder des Transaktionskonzeptes und lieferte in den siebziger Jahren erste wissenschaftliche Veröffentlichungen zu dem Thema.

Dauerhaftigkeit Die Änderungen nach dem Abschluss einer Transaktion müssen vom DBMS rekonstruierbar sein.

Die obigen Regeln werden gemäß ihrer englischsprachigen Initialen[2] als **ACID**-Eigenschaften bezeichnet.[3]

Bemerkung 6.2
Die Atomarität soll garantieren, dass eine Transaktion nicht in einem halbfertigem Zustand verbleibt. Denn bei einem Absturz des Rechners, Stromausfall o. ä. könnte es passieren, dass einige Änderungen ausgeführt werden und andere nicht.

Die Isolation ist insbesondere im Mehrbenutzerbetrieb nützlich. Würde jede Operation sofort wirksam werden, so könnte es sein, dass ein gleichzeitig zugreifender Benutzer mit inkonsistenten Daten weiterarbeitet.

Die Dauerhaftigkeit sichert, dass Daten den Fehlerfall überleben. So kann sich der Benutzer darauf verlassen, dass seine Änderungen Bestand haben, sobald das DBMS diese verarbeitet (Commit) hat.

6.2. Fehlerszenarien

Mit zunehmendem Desaster unterscheidet man Transaktions-, System- und Externspeicherversagen. Im folgenden sollen diese Begriffe erklärt werden.

Definition 6.3 (Transaktionsversagen)
Ein **Transaktionsversagen** liegt vor, wenn eine *einzelne* Transaktion nicht ihr normales Ende erreicht, sondern abbricht.

Bemerkung 6.3
Der Abbruch kann durch den Benutzer, die Anwendung oder durch das DBMS geschehen. Die Gründe für den Abbruch sind in der Regel vielfältig:

- Benutzer hat einen Fehler bemerkt.
- Anwendungsprogramm ist „abgestürzt“.
- Eingabedaten waren inkorrekt und die Anwendung bricht ab.
- Deadlock (Verklemmung)

Definition 6.4 (Systemversagen)
Ein **Systemversagen** liegt vor, wenn das DBMS „abstürzt“, d. h. die gesamte Verarbeitung in der Datenbank wird unterbrochen.

[2] *A*tomicity, *C*onsistency, *I*solation, *D*urability
[3] Die ACID-Eigenschaften wurden erstmalig 1983 von den Forschern Theo Härder und Andreas Reuter in den ACM Computing Surveys formuliert.

Bemerkung 6.4
Bei einem Systemversagen sind alle laufenden Transaktionen und eventuell mehr (bereits abgeschlossene Transaktionen) betroffen. Die Gründe liegen in einem Fehler des DBMS, des Betriebssystems oder einem Hardwarefehler. Daraus resultiert, dass die Inhalte des Hauptspeichers verloren sind. Bereits auf der Festplatte gespeicherte Inhalte bleiben erhalten.

Definition 6.5 (Externspeicherversagen)
Ein **Externspeicherversagen** liegt vor, wenn Daten auf dem Externspeicher verloren sind. Die Datenbank ist ganz oder teilweise zerstört.

Wie wollen nun herausfinden, wie sich die ACID-Eigenschaften dennoch erhalten lassen. Insbesondere die Atomarität und die Dauerhaftigkeit sind von Interesse.

6.3. Klassifikation von Fehlerbehandlungsmaßnahmen

6.3.1. Fehlerbehandlung für Transaktionsversagen

Im Falle des Transaktionsversagens setzt das DBMS alle betroffenen Transaktionen auf ihren Anfangszustand zurück. Änderungen am Zustand der Datenbank werden zurückgenommen. Dies wird als **Rollback der Transaktion** oder **Transaction Rollback** bezeichnet. Die Maßnahme dient dazu, die Atomarität zu garantieren.

Zur Umsetzung protokolliert das DBMS Änderungen am Zustand der Datenbank in Form von Tupel vor der Änderung (**Before Image**) und Tupel nach der Änderung (**After Image**). Diese werden als **Log-Datei** auf die Festplatte geschrieben. Die Tupel vor einer Änderung können dann zum Rollback verwendet werden. Ein Rollback geschieht automatisch durch das DBMS.

6.3.2. Fehlerbehandlung für Systemversagen

Ein DBMS führt verschiedene Transaktionen durch. Bei der Ausführung zweier Transaktionen kommt es zu einem Absturz des Systems. Was ist bei einem Neustart zu-tun?

Die Transaktionen, die während des Absturzes aktiv waren, müssen zur Erhaltung der Atomarität zurückgesetzt werden. Hierzu bedient sich das System, wie oben beschrieben, der Tupel vor Änderung. Unter Umständen müssen bereits abgeschlossene Transaktionen „nachgefahren“, d. h. wieder ausgeführt werden. In der Regel bezeichnet man dies als `REDO`.

Betriebssysteme schreiben Änderungen nicht sofort auf die Festplatte. Vielmehr werden die Daten in einem Puffer gehalten und erst geschrieben, wenn der Puffer voll ist bzw.

wenn das Betriebssystem „Zeit dazu hat“. Daher könnte es sein, dass Transaktionen, die das DMS als abgeschlossen markiert hat, vor dem Crash noch nicht persistent geschrieben wurden. Die After Images helfen, Transaktionen nachzufahren.

Jedoch ist unklar, wie weit in der Vergangenheit mit dem Nachfahren begonnen werden muss. Die todsichere Variante wäre den Startpunkt des DBMS selbst anzusetzen. Jedoch war das System unter Umständen mehrere Tage oder Wochen aktiv und hat viele Transaktionen abgeschlossen. Ein `REDO` wäre in dem Falle sehr aufwändig.

Eine Lösung heißt **Checkpoint** oder **Sicherungspunkt**. In regelmäßigen Abständen wird der Pufferinhalt komplett in die Datenbank geschrieben. Beim Nachfahren muss das DBMS somit nicht über den aktuellsten Sicherungspunkt in die Vergangenheit gehen.

In der Praxis erfolgt das Schreiben sowohl an den Sicherungspunkten wie auch zu anderen nicht vorhersehbaren Zeitpunkten. Verschiedene DBMS schreiben proaktiv Daten auf den Speicher. Die Daten umfassen sowohl abgeschlossene wie noch offene Transaktionen. Daten aus offenen Transaktionen bezeichnet man als **„schmutzige Daten“**. Diese „stehlen“ sich durch das Schreiben davon. Daher heißt die Strategie **Steal-Strategie**.

Write-Ahead-Logging (**WAL**) ist das Schreiben eines Log-Eintrags *vor* dem Datenblock. Dies sichert die Rücksetzbarkeit der Transaktion.

6.3.3. Fehlerbehandlung für Externspeicherversagen

Die Fehlerbehandlung für Externspeicherversagen erfolgt durch das Anlegen von Kopien der Datenbank (**Backup**) sowie durch das Protokollieren aller Änderungen durch After Images.

Im Fehlerfall lädt der Datenbankadministrator die Kopie der Datenbank und lässt darauf alle abgeschlossenen Transaktionen nachfahren. Nach Abschluss befindet sich das DBMS wieder in einem konsistenten Zustand.

6.3.4. Größe der Log-Dateien

Eine Log-Datei kann theoretisch beliebig groß werden. Die natürliche Grenze ist die Größe des Datenspeichers. Unter ungünstigen Umständen reagiert das Betriebssystem mit Stillstand oder Absturz auf vollständig gefüllte Datenspeicher. Eventuell verweigert auch das DBMS seinen Dienst. Daher ist der Zustand nicht erstrebenswert. Wie lässt sich das verhindern?

Die Log-Daten für einen `UNDO` (Before Images) von abgeschlossenen Transaktionen werden nicht mehr benötigt. Denn abgeschlossene Transaktionen können nicht mehr zurückgerollt

werden. Auch Log-Daten für einen `REDO` (After Images), die vor dem letzten Sicherungspunkt angelegt wurden, werden nicht mehr gebraucht. Können die Log-Dateien unter Beachtung der vorstehenden Regeln verworfen werden?

Eine Antwort fällt nicht ganz eindeutig aus. In der praktischen Umsetzung werden die Before und After Images in einer Datei geführt. Das erschwert das selektive Verwerfen. Weiterhin braucht man `REDO`-Dateien für das Externspeicherversagen. Daher können diese nicht so schnell verworfen werden.

DB2 löst dies durch langfristiges Aufheben der `REDO`-Dateien. Sie werden auf einen langsamen Tertiärspeicher kopiert und aufbewahrt. Kurzfristiger benötigte Dateien verbleiben auf dem Speichermedium und werden regelmäßig überschrieben oder gelöscht.

6.3.5. Schreibstrategien eines DBMS

Untenstehend wurde einige Merkmale zur Klassifikation aus [12] herausgegriffen:

force/no force Bei **force** werden die Daten spätestens beim Ende der Transaktion auf die Platte geschrieben. Dagegen geschieht das Schreiben bei **no force** zu einem nicht definierten Zeitpunkt. Produkte bevorzugen die No-Force-Strategie.

steal/no steal Bei **steal** „stehlen“ sich die Daten schon im Verlauf einer Transaktion aus dem Puffer. Schreiben nach dem Ende der Transaktion („saubere“ Daten) heißt **no steal**. In der Praxis ist steal verbreitet.

atomic/not atomic Hierbei geht es um das Schreiben der Daten auf die Datenbank, nicht um das „A“ aus den ACID-Eigenschaften. Punktuelles Wegschreiben der Daten heißt **atomic**. Damit befindet sich der Speicher immer in einem wohldefinierten Zustand. Das kontinuierliche Schreiben heißt **not atomic**.

7. Synchronisation im Mehrbenutzerbetrieb

Auf Datenbanken ist Mehrbenutzerbetrieb unbedingt erforderlich. Man stelle sich nur vor, jede Benutzeranfrage würde nacheinander bearbeitet werden. Im ungünstigsten Falle müsste ein Benutzer sehr lange warteb, bis seine Anfrage beantwortet wird. Der Durchsatz an Daten würde dramatisch sinken. Andererseits führt unkontrollierter Mehrbenutzerbetrieb zu vielfältigen Fehlersituationen und im schlimmsten Falle zu „Chaos“.

7.1. Probleme bei unkontrolliertem Mehrbenutzerbetrieb

Lost-Update-Problem Gegeben seien zwei Transaktionen $T1$ und 2. Die erste Transaktion liest den Wert $A = 30$ aus der Datenbank. Ein Anwendungsprogramm ändert den Wert in 40 und schreibt diesem nach dem Start der Folgetransaktion in die Datenbank. In diesem Moment startet Transaktion $T2$, liest ebenfalls A aus und das Programm ändert den Wert von A in 10. Schließlich wird $A = 0$ in die Datenbank geschrieben. Das bedeutet, die Änderung von Transaktion $T1$ ist verloren gegangen. Bei der oben beschriebenen seriellen Ausführung wäre stets $A = 10$ gewesen.

Inkonsistente Analyse Zwei Transaktionen beginnen gleichzeitig und lesen den Familienstand der Ehefrau aus. Die erste Transaktion ändert nun den Familienstand von verheiratet auf geschieden und macht selbiges mit dem Familienstand des Ehemannes. Anschließend werden die Transaktionen in die Datenbank geschrieben. Während der Zeit wartet die zweite Transaktion. Diese will nun ihrerseits den Familienstand des Ehemannes ebenfalls auf geschieden setzen. Dabei sieht die zweite Transaktion eine vermeintliche Inkonsistenz. Denn die Ehefrau ist als verheiratet markiert, während der Ehemann schon geschieden ist. Die Analyse der zweiten Transaktion bezieht sich auf zwei verschiedene konsistente Zustände der Datenbank.

Phantomproblem Die Integritätsregel einer Datenbank besagt, dass die Anzahl der Konten immer gerade sein muss. Eine Transaktion fängt an, die Konten aller Kunden mit dem Anfangsbuchstaben „A“ auszulesen. Danach fügt eine zweite Transaktion das Konto für Familie Adam ein. Die erste Transaktion fährt mit den Anfangsbuchstaben „B“ und „C“ fort. Nun fügt die zweite Transaktion ein Konto für Herr Zylinski ein und bestätigt die Änderungen mit einem `COMMIT`. Die andere Transaktion liest alle Konten bis zum Anfangsbuchstaben „Z“ ein. Schließlich meldet

sie, dass die Zahl der Konten ungerade ist. Denn das Einfügen von Zylinski hat sie registriert, währenddessen das Einfügen des ersten Kontos unbemerkt an ihr vorüber gegangen ist. Das Phantomproblem ist ein Problem, welches nicht einfach zu lösen ist. Es lässt sich am besten durch eine Sperre der betreffenden Tabelle lösen.

Nicht wiederholbares Lesen Bei diesem Problem handelt es sich um einen Spezialfall der inkonsistenten Analyse. Eine Transaktion bestimmt die Summe aller Kontostände. In der Zwischenzeit erhöht eine andere Transaktion den Kontostand eines Kunden und bestätigt diese. Wenn die erste Transaktion wiederum die Summe aller Kontostände bestimmt, erhält sie ein anderes Ergebnis. Unter Umständen wird bei jeder Ausführung der ersten Transaktion ein anderes Ergebnis geliefert.

Abhängigkeit von nicht freigegebenen Änderungen Eine Transaktion verändert den Wert von A und schreibt diesen in die Datenbank zurück. Eine zweite Transaktion liest den Wert von A aus der Datenbank und zeigt diesen an. Schließlich wird auch hier ein `COMMIT` durchgeführt. Nun erkennt die erste Transaktion einen logischen Fehler und führt einen `ROLLBACK` durch. Der Benutzer, der die zweite Transaktion ausführte, sah einen Zustand den es so nie gab. Dies ist hochgradig unerwünscht. Im obigen Fall ist das nicht durchführbar, da die erste Transaktion die Änderungen schon committet hatte.

Insgesamt lässt sich sagen, dass Synchronisation im Mehrbenutzerbetrieb benötigt wird. Das Sperren ist *eine* Möglichkeit, die Probleme zu lösen.

7.2. Ziel der Transaktionsausführung

Dem Benutzer soll Einbenutzerbetrieb bei Mehrbenutzerbetrieb vorgespiegelt werden. Er soll den Eindruck erhalten, dass er allein auf der Datenbank arbeitet. Trotz Mehrbenutzerbetriebs in der Praxis sollen nur solche Zustände erzeugt werden, die auch bei serieller Abarbeitung der Transaktionsmenge entstehen können. Man bezeichnet diese als Kriterien der Serialisierbarkeit. Eine mögliche Lösung ist das Sperren von Datenbankobjekten (Tupeln, Seiten etc.) *vor* dem Zugriff und dem Befolgen eines **Sperrprotokolls** (**Zwei-Phasen-Sperrprotokoll** und **Freigabeprotokoll**).

7.3. Sperr- und Freigabestrategien

7.3.1. Sperrstrategien

Eine Transaktion kann direkt zu Beginn alle benötigten Datenbankobjekte sperren. Die wird als **Preclaiming** bezeichnet. Eine weitere Strategie ist die Sperrung direkt vor dem Zugriff auf das Objekt. Also das Sperren wird sukzessive durchgeführt.

Beim Preclaiming ist ein optimales Scheduling möglich. Denn alle benötigten Betriebsmittel sind der Transaktion vorab bekannt. Die Gefahr von Verklemmungen (Deadlocks) liegt nicht vor. Beim Einsatz in der Datenbank ist eine solche Strategie in der Regel unrealistisch bzw. nicht implementierbar. Die übliche Strategie ist das sukzessive Sperren. Hier sperrt das DBMS nicht mehr Objekte als unbedingt nötig und auch nicht früher als nötig.

7.3.2. Freigabestrategien

Eine **lange Sperre** liegt dann vor, wenn erst zum Ende der Transaktion die Objekte freigegeben werden. Hingegen bezeichnet eine **kurze Sperre** die Freigabe zu einem früheren Zeitpunkt, nämlich dann, wenn das Objekt nicht mehr benötigt wird.

Das lange Sperren ermöglicht ein isoliertes zurücksetzen und wiederholbares Lesen. Von der „reinen Lehre“ ist das bei Datenbanken zu verwenden. In der Praxis wird das zum Teil aufgeweicht, um die Parallelität zu erhöhen. Bei kurzen Sperren ist eine höhere Parallelität als bei langen Sperren möglich. Daher finden wir das oft in DBMS.

7.4. Sperrgranulate

Der Abschnitt behandelt die Frage, was gesperrt wird. Neben den logischen Datenbankobjekten, wie Datenelementen (Attributwert), Satz (Tupel), Tabelle (Relation) oder der Datenbank selbst sind physische Datenbankobjekte (Seite, Seitenausschnitt, Datenbanksegment) möglich. Es wird ein Kompromiss aus hoher Parallelität, Performance und Implementierungsaufwand gesucht. Kleine Sperrgranulate sind wegen der Parallelität zu bevorzugen. Diese kosten jedoch Performance und sind schwerer umzusetzen. Daher ist von der Betrachtungsweise eher ein gröberes Granulat zu bevorzugen. Derzeit sind Tupel bzw. Indexeinträge die kleinsten Granulate und es gibt eine Hierarchie. Darin folgt dem Tupel eine Relation und darauf ein Segment. Die größte Stufe ist die Datenbank.

7.5. Sperrmodi

In der Regel finden sich in DBMS zwei Sperrmodi, nämlich **Schreib-** und **Lesesperren**. Eine Schreibsperre wird manchmal als X-Sperre[1] oder Exclusive Lock bezeichnet. Diese Sperre kann nur von einer Transaktion für ein Datenbankobjekt gehalten werden. Daher kommt der Name exklusiv. Andere Transaktionen müssen warten. Hingegen können mehrere Transaktionen eine Lesesperre (S-Sperre, Shared Lock) auf einem Objekt besitzen. Dabei ist zu beachten, dass eine Schreibsperre nicht mit einer Lesesperre verträglich ist.

[1] Das "Xßteht für eXecute.

Im Rahmen einer Sperrhierarchie kann es einen so genannten Intention Lock (IS, IX) geben. Dieser drückt aus, dass auf „tieferer Ebene“ mit S oder X gesperrt werden soll. Sperren müssen immer hierarchisch angefordert und entsprechend wieder freigegeben werden. Unterhalb von IS darf wieder IS oder S kommen. Unterhalb von IX darf IX, IS, S, oder X kommen. Unterhalb einer S- oder X-Sperre kommt hingegen nichts mehr. Diese stellen das Ende der Sperranforderungen dar.

Beispiel 7.1 (Sperrhierarchie bzw. Intention Lock)
Die Sperrgranulate seien Relation und Tupel. Eine Transaktion T_i will einige Tupel der Relation R_j ändern. Was ist zu tun? Die einfache Lösung ist die Sperrung der ganzen Relation. Dies verhindert jedoch sämtliche Parallelität. Daher könnten genau die Tupel gesperrt werden, die verändert werden sollen. Was passiert aber, wenn eine parallele Transaktion T_k „unbemerkt“ R_j sperrt? Die Transaktion T_i muss sich an das hierarchische Sperrprotokoll halten. Das heißt, zuerst setzt T_i eine IX-Sperre auf R_j und danach X-Sperren auf die einzelnen zu verändernden Tupel. Die Verträglichkeitsmatrix sieht folgendermaßen aus:

	IS	IX	S	X
IS	+	+	+	-
IX	+	+	-	-
S	+	-	+	-
X	-	-	-	-

Neben den hier vorgestellten Sperrmodi gibt es weitere. Diese spielen im Rahmen der Vorlesung keine Rolle.

7.6. Sperreskalation

Unter Umständen stellen viele kleine Sperren ein Problem dar. Die Größe der Sperrtabelle im Haupt- oder virtuellen Speicher könnte zu stark anwachsen. Weiterhin gibt es eine Vielzahl von Aufrufen an den Sperrverwalter (Lock Manager) zur Anforderung und Freigabe von Sperren. Dies kann die Ausführung der anderen Programme behindern. Einzelne Aufrufe werden zunehmend teurer. Dies lässt sich durch eine geschickte Implementierung eventuell eingrenzen.

Die Sperreskalation bietet eine Lösung. Dabei wird von vielen Tupelsperren auf eine Tabellensperre übergegangen. Die Sperrtabelle wächst nicht weiter, die ganze Tabelle gesperrt ist. Weiterhin müssen keine satzweisen Aufrufe des Lockmanagers erfolgen. Nachteilig sind die stärkere Behinderung infolge des gröberen Granulats. Im ungünstigen Falle kann die Sperreskalation scheitern oder einen Deadlock verursachen.

7.7. Sperr- und Wartegraph

Definition 7.1 (Sperr-/Wartegraph)
Ein **Sperrgraph** bietet eine Information darüber, wer welche Ressource aktuell gesperrt hat. Ein **Wartegraph** bietet Information darüber, wer auf welche Ressource wartet, weil sie aktuell gesperrt ist.

Beispiel 7.2
In der untenstehenden Grafik wartet die Transaktion x auf die Freigabe von Ressource R1 durch Transaktion y. Diese Transaktion wiederum wartet auf Freigabe von Ressource R3 durch Transaktion w. Schließlich wartet Transaktion z auf Freigabe von Ressource R2 durch Transaktion y.

Grafik einbauen

7.8. Optimistic Concurrency Control

Bish betrachteten wir das Sperren von angefassten Datenbankobjekten. Da gab es *erst* eine Sperranforderung und -gewährung. Danach wurden Aktionen durchgeführt. Dies stellt den pessimistischen Ansatz dar. Denn es könnte etwas passieren. Man bezeichnet das als **Pessimistic Concurrency Control** (**PCC**). Diverse Nachteile sind vorhanden:

- Sperranforderung und -freigabe kosten etwas (Aufrufe des Lockmanagers, Einträge in Log-Tabelle)
- eventuell müssen die Transaktionen warten
- Gefahr eines Deadlock
- manchmal wird „unnötig“ gesperrt (bei geringer Konfliktwahrscheinlichkeit)

Diese Nachteile versucht der optimistische Ansatz zu beheben. Dabei werden die Transaktionen zuerst durchgeführt und beim Ende geschaut, ob alles geklappt hat. Das Verfahren hat wenig Kosten zur Laufzeit. Allerdings gibt es Konflikte, wenn sich zwei Transaktionen ins Gehege kommen. Dann muss die fertige Transaktion aufwendig wieder zurückgesetzt werden. Dieser Aufwand ist bei DBMS meist höher als der erzielte Nutzen. Daher wird OCC nicht als Standardmechanismus eingesetzt. Für konfliktarme Anwendungen kann das jedoch das richtige Verfahren sein. Im Idealfall ließe sich das DBMS diesbezüglich anpassen. Derzeit ist das nicht möglich.

7.9. Kurzzeitsperren

Wir haben kennen gelernt, dass Sperren über den Sperrverwalter angefordert und in die Sperrtabelle eingetragen werden. Sollte die Sperre nicht gewährt werden, wartet die

Transaktion. Gegebenenfalls müssen Deadlocks erkannt und aufgelöst werden. Diese Transaktionssperren dienen der Gewährleistung der Isolation im ACID-Prinzip. Solche eine „schwergewichtige“ Sperre ist in der Regel lange (Sekunden oder länger) vorhanden. Im Unterschied dazu stehen die **Kurzzeitsperren**. Sehr oft arbeiten verschiedene Prozesse oder Threads gemeinsam an der Abarbeitung der diversen Aufgaben der Datenbank. Sie können sich wechselseitig ins Gehege kommen, wenn sie gemeinsam interne Datenstrukturen nutzen. Dies betrifft insbesondere Systempuffer, Log-Puffer aber auch die Sperrtabelle selbst. Kurzzeitsperren sind leichgewichtige Sperren für Prozesse. Die Realisierung erfolgt als so genannte **Latches**[2] mittels einer Variable im gemeinsam genutzten Hauptspeicher.

Beispiel 7.3
Ein Prozess analysiert für die Transaktion 1 die Sperrtabelle. Parallel fügt ein anderer Prozess für eine zweite Transaktion etwas in die Sperrtabelle ein und löst unter Umständen Chaos aus.

Latches werden wegen des hohen Behinderungspotenzials nur für sehr kurze Zeiträume (wenige 10 oder 100 Instruktionen) gehalten und sind billig. Bei belegter Kurzzeitsperre kommt es zum Busy Wait, im Gegensatz zu Lazy Wait bei normalen Sperren.

[2] Vom englischen Wort für Schnappschloss

8. SQL zur Anwendungsprogrammierung

Bei den bisherigen Betrachtungen gingen wir davon aus, dass alle SQL-Anweisungen am Bildschirm eingegeben werden. Die Ausgabe der Ergebnisse erfolgt ebenfalls am Bildschirm. Diese Form der Verarbeitung wird als **interaktives SQl** bezeichnet. In der Praxis kommt dieser Fall selten vor. Zumeist arbeitet der Nutzer mit einer Anwendung. Diese erzeugt die entsprechenden SQL-Anweisungen und gibt diese an die Datenbank weiter. SQL ist damit eingebettet in Anwendungsprogrammen, die in einer höheren Programmiersprache geschrieben wurden.

Die Väter von SQL hatten sich dies anders vorgestellt. Der ursprüngliche Name SEQUEL (Structured *English* Query Language) macht dies recht deutlich.

Jedoch sind SQL und die verwendete Programmiersprache unterschiedliche Sprachen. Wie also schreibt man ein Programm, welches beide Sprachelemente enthält? SQL ist als mengenorientierte, deskriptive Sprache entworfen worden. Programmiersprachen sind (oft) prozedural und satzorientiert. Dieses Missverhältnis bezeichnet man als **Impedance Mismatch** oder Object-relational Impedance Mismatch. Im folgenden betrachten wir, wie beide Konzepte zusammengebracht werden.

A. Übungsaufgaben in DBS2

A.1. Übungsblatt 1

Aufgabe 1 (Transaktionen und Geld)
In Datenbanksystemen werden zusammengehörige Änderungen stets zu Transaktionen zusammengefasst. D. h. ein Benutzer ändert nicht einfach (z. B. mit `UPDATE`, `DELETE`, `INSERT`) auf der Datenbank, sondern er macht dies innerhalb einer Transaktion. Das DBMS garantiert ihm, dass die Transaktion entweder komplett oder gar nicht ausgeführt wird (All-or-Nothing-Prinzip, Atomarität), was vor allem für den Fehlerfall — z. B. „Absturz“ des Anwendungsprogramms, das die Transaktion veranlasst hat, oder Stromausfall während der Transaktionsausführung — wichtig ist.
Beispiel (Umbuchung vom Konto Küspert zum Konto Rabinovitch):

1. `BEGIN TRANSACTION`
2. Lese Kontostand Küspert, um zu prüfen, ob Geld vorhanden ist.
3. Ziehe 1000 € von Konto Küspert ab
4. Erhöre Kontostand Küspert um 1000 €
5. `END TRANSACTION`

Diskutieren Sie an diesem Beispiel, was passieren könnte, wenn die Änderungsoperationen nicht in Transaktionsklammern eingeschlossen wären. An welchen Stellen können Fehler auftreten und welche Auswirkungen ergeben sich?

Aufgabe 2 (Checkpoints und Fehlerfall)
Im Fehlerfall muss ein DBMS die ACID-Eigenschaften von Transaktionen sichern. Wir betrachten das folgende Szenario: Grafik einbinden

a) Welche Fehlerbehandlung ist beim Wiederanlauf nach einem Systemversagen (Crash) für die einzelnen Transaktionen nötig? Begründen Sie Ihre Antwort.

b) Versuchen Sie, die Fehlerbehandlung algorithmisch zu beschreiben. Beachten Sie bei Ihren Überlegungen, dass auch unbeendete Transaktionen („schmutzige“) Daten auf die Festplatte schreiben können (z. B. aus Platzmangel im Puffer). Andererseits müssen („saubere“) Daten beendeter Transaktionen auch nicht sofort auf Festplatte in der DB gesichert werden. (Welche beiden Schreibstrategien werden also zugrundegelegt?)

c) Was wären die Implikationen für die Fehlerbehandlung, wenn abweichend von Teilaufgabe b

- geänderte Daten nie vor Transaktionsende auf Festplatte gesichert (in die DB (zurück)geschrieben) werden, oder wenn
- geänderte Daten (spätestens) bei erfolgreichem Transaktionsende automatisch auf Festplatte in die DB geschrieben werden?

Aufgabe 3 (Häufigkeit von Checkpoints)

a) Beim Setzen von Checkpoints im laufenden Datenbankbetrieb spielt die Frequenz (Checkpoint-Intervall-Länge) eine bedeutende Rolle. Man kann Checkpoints selten oder auch häufig setzen. Der DBA bestimmt die Checkpoint-Intervall-Länge. Was spricht für, was gegen die jeweiligen Varianten?

b) Kann im Fehlerfall alles, was vor dem Checkpoint passierte, außer acht gelassen werden? Begründen Sie Ihre Antwort.

Aufgabe 4 (Log-Dateien)

Ein Ziel bei Log-Dateien ist es, diese möglichst klein zu halten. Diskutieren Sie, welcher Abschnitt des Logs in den Fehlerbehandlungsfällen R1—R4 jeweils benötigt wird und welcher Abschnitt nicht benötigt wird.

Aufgabe 5 (Optimierung des Logging, Log-Puffer-Verwendung etc.)

Bei der Durchführung von SQL-Operationen vom System erzeugte Log-Einträge werden nicht sofort auf Sekundärspeicher geschrieben, sondern gelangen zunächst in einen seitenstrukturierten Log-Puffer im Hauptspeicher. (Die Begriffe *Seite* und *Block* seien hier und im Folgenden der Einfachheit halber synonym verwendet.) Beantworten Sie bitte die folgenden Fragen:

a) *Wann* (spätestens) müssen Log-Puffer-Inhalte auf Sekundärspeicher geschrieben werden und aus welchem Grunde geschieht dies jeweils?

b) Wir nehmen an, dass eine Transaktion mehrere Tupel ändert und dann mit Commit beendet wird. Wie viele I/O-Operationen pro Transaktion(s-Commit) werden mindestens benötigt und wozu? Läßt sich diese Zahl an I/O-Operationen weiter reduzieren / durch welches Verfahren ist dies möglich? Was ist der (kleine) Nachteil hiervon?

c) Wann ist eine Transaktion „wirklich (erfolgreich) beendet“ / wann kann der Anwender von Durability-Eigenschaft der Änderungen ausgehen?

d) Was unterscheidet Datenbank-I/O-Operationen in Performance-Dingen prinzipiell von Log-I/O-Operationen?

e) Was ist das Hauptproblem von no-steal?

f) Welche Vorteile/Nachteile hätte man bei Verwendung mehrerer Logs?

Aufgabe 6 (Verschiedenes)

a) Was machen die beiden ACID-Erfinder heute, d. h. wo sind sie tätig? **foo**.

b) Wer war der Vater der Transaktionsthematik? Was macht er heute? Wo war er zuletzt tätig?

A.2. Übungsblatt 2

Aufgabe 7 (Zusammenspiel von DB-Puffer und Log-Puffer)
Gegeben seien (alles stark vereinfacht und „verkleinert“) die DB-Seiten $S1, S2, S3$ und $S4$. Die Seitengröße betrage 4 kB. Der Log-Puffer habe eine Kapazität von einer Seite. Der DB-Puffer kann max. 2 Seiten aufnehmen. Des Weiteren setzen wir eine konstante Tupellänge von 1 kB und eine Log-Eintrags-Größe von 2 kB voraus.

Als Seitenverdrängungsstrategie des DB-Puffers nehmen wir *FIFO* (*first in first out*) an, d. h., verdrängt wird bei Bedarf diejenige Seite, die als erste in den Puffer geladen wurde. Wir setzen ferner voraus, dass eine Seite nur dann auf Platte ausgeschrieben wird, wenn dies (aus irgendeinem Grund) notwendig ist, also kein Einsatz von proaktiven Page Cleaner. Des Weiteren werde eine Seite nur dann in den DB-Puffer geladen, wenn auf diese Seite zugegriffen wird (also kein „vorausschauendes“ Puffern, Prefetching). Für den DB-Puffer werde *STEAL* sowie *NO FORCE* angenommen.

Betrachten Sie die folgende Sequenz von Operationen und ermitteln Sie nach jedem Schritt jeweils den Inhalt von Log-Puffer und DB-Puffer. Beide Puffer seien zu Beginn leer.

1. Lies Tupel $t1$ in Seite $S1$.
2. Ändere Tupel $t1$.
3. Lies Tupel $t2$ in Seite $S2$.
4. Erneuter Lesezugriff auf Tupel $t2$.
5. Lies Tupel $t3$ in Seite $S3$.
6. Ändere Tupel $t3$.
7. Commit.
8. Lies Tupel $t4$ in Seite $S4$.
9. Ändere Tupel $t4$.
10. Lies Tupel $t5$ in Seite $S4$.
11. Ändere Tupel $t5$.
12. Lies Tupel $t6$ in Seite $S4$.
13. Ändere Tupel $t6$.
14. Commit.

Aufgabe 8 (Fehlerbehandlung nach Externspeicherversagen)

a) Stellen Sie sich vor, Sie haben Ihr Studium erfolgreich abgeschlossen und arbeiten jetzt als DBA in einem großen Jenaer Unternehmen. Es ist Montagmorgen, 5:30 Uhr und Sie kommen gerade fröhlich pfeifend ins Büro. Noch bevor Sie Ihre Jacke ausgezogen und im Kleiderschrank verstaut haben, müssen Sie zu Ihrem blanken Entsetzen feststellen, dass eine Festplatte das letzte Wochenende nicht „überlebt“ hat. Dabei handelt es sich ausgerechnet um die Platte, auf der die Daten der wichtigsten Datenbank liegen (bzw. gelegen haben). Zum Glück haben Sie aber letzte Woche Freitag eine Vollkopie Ihrer Datenbank erstellt. Was ist jetzt zu tun, d. h. welche Schritte (Aktionen) müssen Sie durchführen bzw. veranlassen?

b) Wie bei a), aber wir nehmen an, dass die letzte Vollkopie der Datenbank sieben Tage alt ist und danach mehrfach inkrementelle Kopien der Datenbank erstellt wurden.

Aufgabe 9 (Inkrementelles Backup)

Wir greifen nun den Sachverhalt der Teilaufgabe 2b) nochmals an einem stark vereinfachten und verkleinerten Beispiel auf. Wie nehmen dabei an, die zu betrachtende Datenbank bestehe aus den Blöcken $1, 2, \ldots, 9$ und 10. Die Blockgröße betrage 4 kB. Ferner unterstellen wir, dass während der letzten Woche die folgenden Ereignisse stattgefunden haben:

1. *(letzte Woche) Montag, 7:00 Uhr*: Komplett-Backup der DB wurde erstellt.
2. *im Laufe des Montags*: Änderungen an den Blöcken 5 und 7.
3. *Dienstag früh*: Änderungen an den Blöcken 2 und 7.
4. *Dienstagmittag*: Inkrementelles Backup der DB wurde erstellt.
5. *Dienstagabend*: Änderungen an den Blöcken 5 und 7.
6. *Mittwoch früh*: Änderungen an den Blöcken 7 und 8.
7. *Mittwochmittag*: Inkrementelles Backup der DB wurde erstellt.
8. *Mittwochabend*: Änderungen an den Blöcken 2 und 9.
9. *Donnerstag früh*: Änderungen an den Blöcken 7 und 9.
10. *Donnerstagmittag*: Inkrementelles Backup der DB wurde erstellt.
11. *Donnerstagabend*: Änderungen an den Blöcken 1 und 4.
12. *Freitag*: Wegen Feierlichkeiten zum Firmenjubiläum kein Zugriff auf die DB.
13. *Samstag, 2:59 Uhr*: Die Platte geht kaputt.
14. *Sonntag*: Niemand hat gemerkt, dass die Platte kaputt ist.

a) Welche Blöcke wurden bei den einzelnen Backups jeweils gesichert?

b) Wie groß ist das Gesamt-Datenvolumen (in kB) aller Backups?

c) Wie groß wäre das Gesamt-Datenvolumen aller Backups, wenn anstatt der inkrementellen Backups stets Komplett-Backups durchgeführt worden wären?

d) In welcher Reihenfolge müssen die Backups wieder eingespielt werden?

e) Wie groß ist das Gesamt-Datenvolumen (in kB) aller einzuspielenden Backups?

f) Wie groß wäre das Gesamt-Datenvolumen aller einzuspielenden Backups, wenn anstatt der inkrementellen Backups stets Komplett-Backups durchgeführt worden wären?

g) Wie oft wurde der Block 7 wieder eingespielt?

h) Wie oft wäre der Block 7 wieder eingespielt worden, wenn anstatt der inkrementellen Backups stets Komplett-Backups durchgeführt worden wären?

i) Könnte man im konkreten Fall auf das Wiedereinspielen eines oder mehrerer inkrementeller Backups verzichten?

j) Wie könnte man für den Backup-Fall Vorkehrungen treffen, damit die inkrementellen Backups nicht (zeitaufwendig) der Reihe nach eingespielt werden müssen?

A.3. Übungsblatt 3

Aufgabe 10 (Sperreskalation)
Gegeben seien die in Abbildung A.1 dargestellte (verkürzte) Sperrgranulathierarchie und die zwei Transaktionen T1 und T2 mit den ebenfalls dargestellten Operationsfolgen (R=read, W=write). Wenn einer Transaktion mehr als 3 Sperren auf Tupelebene innerhalb einer Seite gewährt werden, so werden diese auf die Seitenebene eskaliert (vereinfachende Annahme, in der Realität wird natürlich nicht so „simpel“ – und früh – eskaliert).

a) Beschreiben Sie den Vorgang der Sperreskalation am Beispiel der Transaktion T1 (ohne Berücksichtigung von T2) und erläutern Sie die Vorteile dieses Verfahrens. Zeichnen Sie die Sperranforderungen/-gewährungen in den Baum ein. Wie viele Sperren hält T1 am Ende? Wie viele wären es ohne Sperreskalation?

b) Betrachten Sie nun T1 und T2 mit der gegebenen zeitlichen Operationsabfolge (Nebenläufigkeit). Zeichnen Sie wiederum die Sperranforderungen/-gewährungen in den Baum ein. Welcher schröckliche Effekt tritt dabei auf? Welche Möglichkeiten des DBMS-Verhaltens gäbe es dabei prinzipiell? Diskutieren Sie an diesem Beispiel Nachteile der Sperreskalation.

Aufgabe 11 (Transaktionen und Zweiphasensperrprotokoll (2PL))
Der Mehrbenutzerbetrieb erfordert Maßnahmen zur Synchronisation von Transaktionen, um die ACID-Eigenschaften abzusichern. Prüfen Sie durch Beanwortung folgender Fragen Ihr Wissen auf diesem Gebiet.

a) Was versteht man unter Serialisierbarkeit und unter dem Zweiphasensperrprotokoll?

b) Wie beginnen und wie enden Transaktionen in SQL?

c) In der Vorlesung wurden verschiedene Sperr- und Freigabestrategien des Zweiphasensperrprotokolls angesprochen. Eine davon ist im Hinblick auf die ACID-Eigenschaften bedenklich. Welche ist es und welcher Sperrmodus (S, X) bereitet Probleme? Erläutern Sie an einem Beispiel, was passieren kann.

Aufgabe 12 (Serialisierbarkeit)
Die Transaktionen T1, T2 und T3 sollen auf einem Wert A in einer Datenbank folgende Operationen ausführen:

T1: Addiere 1 zu A.

T2: Verdopple A.

T3: Gebe A aus und setze A danach auf 1.

Die Transaktionen sind intern wie folgt strukturiert:

a) Angenommen, die drei Transaktionen werden konkurrierend ausgeführt, also (annähernd) gleichzeitig initiiert. *A* habe zu Beginn den Wert 0. Geben Sie alle möglichen *korrekten* Resultate der Ausführung an. (Welche Resultate ergäben sich übrigens bei einem Anfangswert von 2010?)

b) Geben Sie eine Ausführungsreihenfolge der sechs Operationen an, die für den Anfangswert 0 ein korrektes Resultat liefert (eines der Resultate von Teilaufgabe a), aber nicht serialisierbar ist.

c) Gibt es eine Ausführungsreihenfolge, die serialisierbar ist, aber nicht gewählt werden kann, wenn alle Transaktionen dem Zweiphasensperrprotokoll genügen sollen? Erläutern Sie Ihre Antwort.

d) Wieviele mögliche Ausführungsreihenfolgen (schedules) der sechs Datenbankoperationen gäbe es *ohne* Sperren? Versuchen Sie, eine allgemeine Berechnungsvorschrift für das Problem zu finden. Was sagt deren Größenordnung uber Möglichkeiten aus, Scheduling-Fragen problemabhängig zur Laufzeit zu entscheiden und zu optimieren?

Aufgabe 13 (Sperrhierarchien)
In der Vorlesung wurden Sperrhierarchien und Intention Locks besprochen. Betrachten Sie die Hierarchie physischer Datenbankobjekte in Abbildung A.2. Wir betrachten nun die Sperranforderungen verschiedener Transaktionen. Kennzeichnen Sie in jeder Teilaufgabe die Datenbankobjekte in der Form (T_i , SP), wobei SP die Art der Sperre (z. B. IS oder X) und T_i die zugehörige Transaktion kennzeichnet. Geben Sie zusätzlich die Reihenfolge der Sperrvergabe mit an.

a) T_1 will die Seite p_1 exklusiv sperren.

b) T_2 will die Seite p_2 mit einer S-Sperre belegen.

c) T_3 will den Table Space a_2 mit X sperren.

d) T_4 will den Datensatz s_3 exklusiv sperren.

e) T_5 will eine S-Sperre auf s_5 erwerben.

f) Da T_4 und T_5 nicht alle Sperranforderungen erfüllt bekommen, sind sie blockiert. Liegt eine Verklemmung vor?

g) Wenn man in Abbildung A.2 noch Tabellen als weitere Objektebene aufnehmen wollte, wo käme dies in den Baum rein?

h) Welche möglichen weiteren Objektebenen wurden in der Vorlesung angesprochen?

- Direkt unterhalb der Tabellenebene
- „und noch darunter liegend“?

Die Sperranforderungen der beiden folgenden Transaktionen können nicht mehr vollständig erfüllt werden (warum nicht?). Kennzeichnen Sie nicht gewährte Sperren hier durch Unterstreichen.

A.4. Übungsblatt 4

Aufgabe 14 (Sperr-/Wartegraph und Sperrtabelle)
Gegeben sei die Hashfunktion $h(Rx) \equiv x \mod 3$ sowie folgender Sperr-/Wartegraph:

Graf zeichnen

a) Stellen Sie die zugehörige Sperrtabelle auf. Beschränken Sie sich dabei auf S und X als zu betrachtende Sperrmodi.

b) Gibt es verschiedene Möglichkeiten („Freiheitsgrade“), die Sperrtabelle zu erstellen? Welche Informationen enthält sie gegebenenfalls, die der Sperr-/Wartegraph nicht enthält?

Aufgabe 15 (Wartebeziehungen)
Sie haben in der Vorlesung Sperrtabellen und Sperr-/Wartegraphen kennen gelernt. Die Information, welche Transaktion auf welche Transaktion wartet, kann aber auch mittels sogenannter Bitmatrizen dargestellt werden. Gegeben sei folgende Bitmatrix (Tabelle A.1), die angibt, welche Transaktion auf welche wartet. Beispielsweise besagt die „1“ am Schnittpunkt der Zeile „TA1“ mit der Spalte „TA2“, dass TA1 auf TA2 wartet. (Genauer gesagt wartet TA1 auf die Freigabe einer Ressource, die von TA2 gesperrt ist.)

a) Stellen Sie den zugehörigen Sperr-/Wartegraphen auf! (Die Ressourcennamen mögen hier keine Rolle spielen und sollen deshalb weggelassen werden.)

b) Können Sie die Bitmatrix in die Darstellung als Sperrtabelle gemäß Vorlesungsfolie 38transformieren?

Verweis einbauen

c) In der obigen Bitmatrix wird nur direktes Warten berücksichtigt. Kann man an dieser Bitmatrix (direkt) ablesen, ob ein Deadlock vorliegt?

d) Wenn eine Transaktion TAi auf die Transaktion TAj wartet, die wiederum auf die Transaktion TAk wartet, so wartet TAi indirekt auf TAk. Geben Sie nun eine Bitmatrix an, die auch indirektes Warten berücksichtigt (transitive Hülle).

e) Woran erkennt man an dieser neuen Bitmatrix, ob ein Deadlock vorliegt?

f) Woran erkennt man im Sperr-/Wartegraphen (1. Teilaufgabe), ob ein Deadlock vorliegt?

g) Liegt im konkreten Fall ein Deadlock vor? Wenn ja, wie kann dieser aufgelöst werden?

h) Woran erkennt man in der Bitmatrix, ob eine Transaktion aktuell arbeitsfähig ist, d. h. nicht auf die Freigabe einer Ressource wartet?

i) Woran erkennt man im Sperr-/Wartegraphen, ob eine Transaktion aktuell arbeitsfähig ist?

j) Welche Transaktionen sind im konkreten Fall aktuell arbeitsfähig?

k) Welche Transaktionen werden arbeitsfähig, wenn TA6 die von ihr belegten Ressourcen wieder freigibt?

l) Angenommen, die Transaktion TA7 gibt ihre Ressourcen wieder frei. Welche weiteren Transaktionen müssen ihre Ressourcen ebenfalls freigeben, bevor TA6 arbeitsfähig wird?

Grafik einbauen

Aufgabe 16 (Sperrtabelle und Sperr-/Wartegraph)
Gegeben sei die folgende (natürlich stark verkleinert gewählte) Sperrtabelle.

Hinweis zur Implementatierung: Im Primärbereich der Hashtabelle (links im Bild, also in der eigentlichen Hashtabelle) befinden sich nur Pointer, sie ist somit sehr kompakt speicherbar. (In der Realität würde man natürlich in einem viel größeren Bereich als nur {0,1} hashen.) Auch die Sperrkontrollblöcke (Ri im Bild) bestehen im Grunde nur jeweils aus drei Pointern, wie man sieht, plus dem Ressourcennamen bzw. intern als ID/Nummer dargestellt (es werden also nicht „die Ressourcen selbst“ hier gespeichert natürlich). Kompaktheit der Sperrtabelle insgesamt ist wichtig, weil sie u vom DBMS im *Hauptspeicher* gehalten werden muss und sich diese rare Ressource mit anderen wichtigen DBMS-Ressourcen teilt (insb. Systempuffer (Cache), Log-Puffer, Sort-Heaps etc.).

a) Sind die angegebenen Sperrmodi zulässig? Wenn nein, wo liegt der Fehler?

b) Stellen Sie den zugehörigen Sperr-/Wartegraphen auf.

c) Liegt im konkreten Fall ein Deadlock vor? Wenn ja, welcher?

d) Angenommen, Sie würden die Wartebeziehungen in einer Bitmatrix darstellen, die außer direktem Warten auch indirektes Warten berücksichtigt. Wieviele Einsen enthielte die Hauptdiagonale dieser Bitmatrix? (Hinweis: Sie brauchen die Bitmatrix *nicht* aufschreiben. Die Hauptdiagonale verläuft von oben links nach unten rechts.)

Aufgabe 17 (Latches)
Damit sich mehrere DBMS-Server-Prozesse oder -Threads z. B. beim Zugriff auf die Sperrtabelle nicht gegenseitig „ins Gehege kommen“, werden **Kurzzeitsperren** genutzt. Die Realisierung der Kurzzeitsperren erfolgt als **Latches** (Schnappschlösser) beispielsweise mit folgender (vereinfachter) Programmlogik:
Falls der Latch frei ist, wird der auf belegt gesetzt und auf der Sperrtabelle gearbeitet. Am Ende der Bearbeitung wird der Latch auf frei gesetzt. Sollte der Latch anfangs belegt sein, wird gewartet (ggf. Busy Wait).

Welche Probleme können auftreten, wenn mehrere Prozesse parallel auf einer Maschine arbeiten wollen und das if/then *nicht* atomar über *eine* Maschinenistruktion realisiert ist?

A.5. Übungsblatt 5

Aufgabe 18 (Einbettung)
Datenbanksprachen (z. B. SQL) werden in höhere Programmiersprachen (z. B. Java, C++, teils gar COBOL, PL/1, schröcklich) eingebettet. Weitere anschauliche Details uber die Kopplungsarten von Programmier- und Datenbanksprachen finden sich in dem zur Vorlesung ausgehändigten Informatik-Spektrum-Papier von Karl Neumann.

a) Erläutern Sie, *warum* eine solche Einbettung überhaupt vorgenommen wird.

b) Welche Probleme treten bei der Einbettung auf?

c) Welche prinzipiellen Ansätze für eine Programmierspracheneinbettung gibt es?

d) Welchem dieser Ansätze ist ESQL zuzuordnen?

Aufgabe 19 (Impedance Mismatch und Cursor)
Cursor spielen in eingebettetem SQL eine wichtige Rolle. Testen Sie Ihr umfangreiches Wissen durch Beantwortung der folgenden Fragen.

a) Was versteht man unter *Impedance Mismatch*?

b) Was versteht man in diesem Zusammenhang unter *Cursor*? Beschreiben Sie das Wesentliche am Cursor-Konzept.

c) An was (an welches Konstrukt) wird ein Cursor gebunden?

d) Was passiert bei `DECLARE CURSOR`?

e) Was passiert bei `OPEN CURSOR`?

f) Was passiert bei `FETCH`?

g) Wann wird die `SELECT`-Anweisung ausgeführt und wo steht der Cursor nach der Ausführung? Warum steht er da und wie lange?

h) Wird bei ESQL stets ein Cursor benötigt?

Aufgabe 20 (`UPDATE`-Operation und Cursor)
Cursor spielen in SQL nicht nur beim lesenden Zugriff aufs Anfrageergebnis eine Rolle. Auch die SQL-`UPDATE`-Operation kann Bezug auf den Cursor nehmen.

a) Wie nennt man diese Form des `UPDATE`?

b) Erläutern Sie die Form des `UPDATE` an einem Beispiel.

c) Können Sie die Form des `UPDATE` durch Eingabe von Ad-Hoc-Anweisungen am Bildschirm testen?

d) Wie unterscheidet sich diese Form des `UPDATE` vom „gewöhnlichen“ `UPDATE` (also dem `UPDATE` ohne Bezugnahme auf den Cursor)?

A.6. Übungsblatt 6

Aufgabe 21 (ESQL)
Die Tabelle `Kunde` besitzt die Spalten `KdNr`, `Name` und `Umsatz`. In dieser Tabelle sind die Kundennummern, Namen und bisherigen Umsätze aller Kunden eines bekannten Unternehmens gespeichert, welches aus Geheimhaltungsgründen hier nicht genannt werden darf (militärisch-industrieller Komplex). Es soll nun ein Anwendungsprogramm entwickelt werden, das sich vom Benutzer einen Betrag eingeben lässt und dann alle Kunden (KdNr, Name und Umsatz) ausgibt, deren Umsatz a den eingegebenen Betrag übersteigt (neudeutsch „Key Accounts“).

a) Muss in diesem Beispiel das Cursor-Konzept genutzt werden? Warum bzw. warum nicht?

b) Skizzieren die wesentlichen „datenbankrelevanten“ Teile des entsprechenden Programms. Sie können hierbei Pseudocode verwenden.

c) Beschreiben Sie die Funktion der ESQL-Anteile Ihres Programms.

Aufgabe 22 (Lehrstuhl-Exkursion)
Bei Exkursionen kann es für die besuchten Unternehmen wichtig sein, aus welchen Städten die Exkursionsteilnehmer stammen (Jena, Rio de Janeiro, Pjöngjang etc.). In der Exkursions-Datenbank sollen deshalb sowohl Informationen uber die Exkursionsteilnehmer (Name, Vorname, Matrikelnummer, Wohnort) als auch Angaben zu ihren Wohnorten (Name, Einwohnerzahl, Bundesland) gespeichert werden. Da auf Teilnehmer- und Städteinformationen i. d. R. gemeinsam zugegriffen wird, erfolgt die Speicherung der Daten in einer Tabelle. Diese Tabelle besitzt die Spalten `Name`, `Vorname`, `MatNr`, `Wohnort`, `Einwohnerzahl` und `Bundesland`. `MatNr` ist Primärschlüssel.

a) Überlegen Sie sich geeignete Beispieldaten (Beispieltupel) und demonstrieren Sie, wie es zu Einfüge-, Lösch- und Änderungsanomalien kommen kann.

b) Wie lassen sich derartige Anomalien verhindern? Geben Sie eine bessere Modellierung an.

Aufgabe 23 (Funktionale Abhängigkeiten)
Gegeben sei das Relationenschema Angestellter(Name, geboren, PLZ, Ort, Straße, SV, AbtNr, AbtName, APNr) für Angestellte. Ein Angestellter unserer Firma hat also eine Sozialversicherungsnummer, dazu sind Name, Geburtsdatum und Adresse gegeben. Man beachte, dass das Geburtsdatum auch aus der Versicherungsnummer herzuleiten ist. Für einen Angestellten ist zudem die Abteilung an gegeben (AbtNr, AbtName), seine Personalnummer (APNr) ist aber nur innerhalb einer Abteilung eindeutig. Ferner setzen wir (nicht ganz realitätsnah) voraus, dass der Ort eindeutig durch die Postleitzahl bestimmt wird.

a) Identifizieren Sie die funktionalen Abhängigkeiten, die Ihrer Meinung nach in Angestellter bestehen. Beschränken Sie sich dabei auf die „grundlegenden“ funktionalen Abhängigkeiten, d. h. auf solche, die sich nicht aus anderen von Ihnen gefundenen funktionalen Abhängigkeiten ableiten lassen.

b) Was versteht man unter „`voller` funktionaler Abhängigkeit“ (siehe Vorlesungsfolie)? Erläutern Sie dies kurz an einem Beispiel.

c) Finden Sie die Schlüssel(kandidaten) des Relationenschemas.

d) Führt das Schema zu redundanter Datenhaltung? Begründen Sie Ihre Antwort.

e) Untersuchen Sie das Schema auf Einfüge-, Lösch- und Änderungsanomalien. Geben Sie entsprechende Beispiele an.

f) Versuchen Sie, das Relationenschema so in mehrere kleinere Schemata aufzuspalten (zu normalisieren), dass die Anomalien von Teilaufgabe e) verschwinden.

g) Geben Sie für die entstehenden Schemata die funktionalen Abhängigkeiten an.

h) Eine zu starke Normalisierung kann in einigen Fällen unpraktikabel sein, weil Zusammenhänge auseinandergerissen werden (vgl. die entsprechende Diskussion/Darstellung in der Vorlesung). Sehen Sie ein solches Beispiel in Ihrer Aufteilung von Teilaufgabe f)? Begründen Sie Ihre Antwort.

i) Geben Sie eine Alternative zu Ihrer Modellierung an, die das in Teilaufgabe h) angesprochene Problem behebt.

Hinweis: Beachten Sie bei Ihrer Argumentation, dass die Diskussion von Normalformen auf der Ebene des Relationenmodells („reine Codd'sche Lehre“) stattfindet, nicht auf der von SQL. Konzepte wie NULL sind hier nicht bekannt; alle Attribute von konkreten Tupeln haben auch Werte. (Codd hat zeitlebens gegen das Erlauben von NULL-Werten angekämpft — wegen damit verbundener Probleme bei Modellierung und Sprache.)

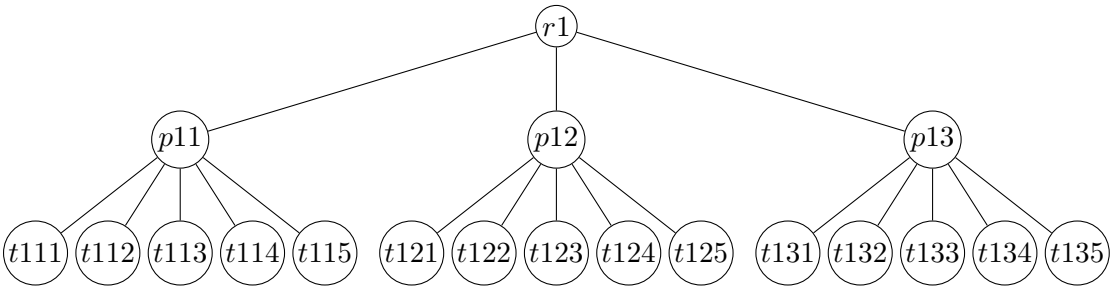


Abbildung A.1.: Hierarchie von Datenbankobjekten

T1	T2	T3
R1: Retrieve A into $t1$ $t1 \coloneqq t1 + 1$ U1: Update A from $t1$	R2: Retrieve A into $t2$ $t2 \coloneqq t2 \star 2$ U2: Update A from $t2$	R3: Retrieve A into $t3$ Display $t3$ U3: Update A from 1

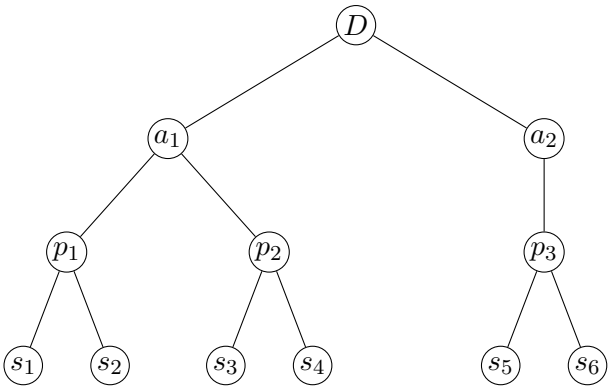


Abbildung A.2.: Hierarchie physischer Datenbankobjekte

	TA1	TA2	TA3	TA4	TA5	TA6	TA7	TA8	TA9
TA1	0	1	0	0	0	0	0	0	1
TA2	0	0	1	1	0	0	0	0	0
TA3	0	0	0	0	1	1	0	0	0
TA4	0	0	0	0	0	0	0	0	0
TA5	0	0	0	0	0	1	0	0	0
TA6	0	0	0	0	0	0	1	1	0
TA7	1	0	0	1	0	0	0	0	0
TA8	0	0	0	0	0	0	0	0	1
TA9	0	0	0	0	0	0	0	0	0

Tabelle A.1.: Bitmatrix

Literaturverzeichnis

[1] Gunter Saake, Kai-Uwe Sattler, Andreas Heuer: Datenbanken: Konzepte und Sprachen. mitp-Verlag, Bonn, 3. Auflage, 2007

[2] Andreas Heuer, Gunter Saake, Kai-Uwe Sattler: Datenbanken kompakt: Einstieg, Datenbanken im Web, CGI, ASP, Content Management, Tuning. mitp-Verlag, Bonn, 2. Auflage, 2003

[3] Stefan Lang, Peter C. Lockemann: Datenbankeinsatz. Springer-Verlag, Berlin, 1. Auflage, 1995

[4] Werner Kiessling, Gerhard Köstler: Multimedia-Kurs Datenbanksysteme, m. CD-ROM. Springer-Verlag, Berlin, 1. Auflage, 1998

[5] Karl Neumann: Datenbanktechnik für Anwender. Hanser Fachbuch, 1996

[6] Alfred Moos, Gerhard Daues: Datenbank-Engineering: Analyse, Entwurf und Implementierung relationaler Datenbanken mit SQL. Vieweg Verlagsgesellschaft, 2. Auflage, 1997

[7] Günter Matthiessen, Michael Unterstein: Relationale Datenbanken und SQL. Addison-Wesley, München, 4. Auflage, 2007

[8] Donald Andrew Jardine: The ANSI/SPARC DBMS Model. North-Holland Pub. Co., Amsterdam, 1977

[9] Theo Härder, Erhard Rahm: Datenbanksysteme. Konzepte und Techniken der Implementierung. 2. Auflage. Springer-Verlag, 2001

[10] Gunter Saake, Andreas Heuer, Kai-Uwe Sattler: Datenbanken: Implementierungstechniken. mitp-Verlag, 2005.

[11] Rick Long, Mark Harrington, Robert Hain, Geoff Nichols: IMS Primer. IBM, 2000. http://www.redbooks.ibm.com/redbooks/pdfs/sg245352.pdf

[12] Theo Härder, Andreas Reuter: Principles of Transaction-oriented Database Recovery. ACM Computing Surveys, Vol. 15, No. 4, Dezember 1983.

Index